HOCHSCHULE FURTWANGEN UNIVERSITY HFU

Masterthesis

im Studiengang

Informatik Master

Generischer Cloud Ressourcen Broker zur Bereitstellung und Konfiguration von VMs im Bezug auf die Analyse und Verarbeitung problemspezifischer Datensätze

Referent	:	Prof. Dr. Christoph Reich
Koreferent	:	M.Sc. Dirk Hölscher
Vorgelegt am	:	31.08.2017
Vorgelegt von	:	Philipp Ruf Matrikelnummer: 253302 Merzhauserstraße 28, 79100 Freiburg Philipp.Ruf@hs-furtwangen.de

Abstract

Die Vielfalt von heutzutage auftretenden Datenstrukturen schafft Bedarf für individuell abgestimmte Analyseplattformen. Dabei benötigte Ressourcen sind vom jeweiligen Anwendungsfall abhängig. Diese Arbeit diskutiert Broker für die Virtualisierung der verarbeitenden Anwendungen, welche durch ein abstrahiertes Dashboard bedient werden. Eine Domain Specific Language ermöglicht die Generierung eines Grundgerüsts entsprechender Komponenten, die mit individueller Logik anzureichern sind. Die beschriebene Architektur bezieht sich zu großen Teilen auf den Umgang mit den flexiblen Eingangsdaten von virtualisierten Verarbeitungsplattformen.

The variety of todays data structures creates demands for individually adapted analysis platforms. The required resources for this achievement always depend on the particular scenario. This work debates broker architectures for managing virtualized processing applications inside a Dashboard. A Domain Specific Language enables the generation of the accordingly component skeletons, which must be enriched with individual logic. In large part, the described architecture refers to the usage with flexible input data and the configuration of virtualized processing platforms.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AAS Audit Agent System

ACO Ant Colony Optimization

AD Active Directory

API Application Programming Interface

AST Abstract Syntax Tree

BEAM Bjorn Gustavsson/ Bogumil Hausman Erlang Abstract Machine

CEP Complex Event Processing

CFS Cassandra File System

CIM Computation Independent Model

CLI Command Line Interface

CMS Cloud Management System

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CSC Cloud Service Custumer

CSP Cloud Service Provider

DBMS Database Management System

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DSL Domain Specific Language

EBNF Extended Backus–Naur form

EC2 Elastic Compute Cloud

EO Earth Observation

FIFO First-In-First-Out

GPL General Purpose Language

GUI Graphical User Interface

HDFS Hadoop Distributed File System

HOT Heat Orchestration Template

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IDL Interface Definition Language

IKE Internet Key Exchange

IP Internet Protocol

IPSec Internet Protocol Security

JADE Java Agent DEvelopement

JSON JavaScript Object Notation

JVM Java Virtual Machine

KPI Key Performance Indicator

LDAP Lightweight Directory Access Protocol

LXDE Lightweight X Desktop Environment

M2C Model to Code

M2M Model to Model

MAC Media Access Control

MDA Model Driven Architecture

MDSD Model Driven Software Development

ML2 Modular Layer 2

MOF Meta Object Facility

NaaS Network as a Service

NAT Network Address Translation

NFS Network File System

NIST National Institute of Standards and Technology

OCL Object Constraint Language

OLAP Online Analytical Processing

OLTP Online Transaction Processing

OS Operting System

OSD Object Storage Devices

OSI Open Systems Interconnection

OTP Open Telecom Platform

OVS Open vSwitch

PaaS Platform as a Service

PAM Pluggable Authentication Modules

PID Process ID

PIM Platform Independent Model

PSE Parameter Sweep Experiments

PSM Platform SpecificModel

QoE Quality of Experience

QoS Quality of Service

RADOS Reliable Autonomic Distributed Object Store

RBD RADOS Block Device

RDBMS Relational Database Management System

RDD Resilient Distributed Dataset

RDF Resource Description Framework

RDP Remote Desktop Protocol

REST REpresentational State Transfer

RMI Remote Method Invocation

RPC Remote Procedure Call

SaaS Software as a Service

SCRB Semantic Cloud Resource Broker

SLA Service Level Agreement

SOA Service Oriented Architecture

SPICE Simple Protocol for Independent Computing Environments

SPOF Single Point of Failure

SQL Structured Query Language

SSH Secure SHell

SSHFS Secure SHell File System

TCP Transmission Control Protocol

URL Uniform Resource Locator

VM Virtual Machine

VNC Virtual Network Computing

VPN Virtual Private Network

VPNaaS VPN as a Service

XML eXtensible Markup Language

XSD XML Schema Definition

YARN Yet Another Resource Negotiator

1. Einleitung

Heutzutage existiert eine Vielzahle an cloudbasierten Systemen und Diensten, die auf unterschiedlichen Infrastrukturen und an verschiedenen Standorten ausgeführt werden. Die über den gesamten Globus verteilten Rechenzentren stellen eine flexible und skalierbare Infrastruktur bereit, welche auf dem Konzept der Virtualisierung von physischen Ressourcen beruht. Grundlegende Technologien für diesen Trend sind mittlerweile ausgereift und wurden in den letzten Jahren in verschiedensten Unternehmungen im privaten und geschäftlichen Bereich produktiv eingesetzt [BKNT11].

Die Anwendung dezentraler und jederzeit über das Internet verfügbarer Infrastrukturen, Plattformen oder Anwendungen verschiedener Anbieter ist die Grundlage verschiedener aktueller Geschäftsmodelle. Derartige Angebote beziehen sich in den meisten Fällen auf eine bestimmte Problemdomäne und werden dem Verbraucher prinzipiell als Dienst angeboten [MG11].

Der Betrieb eines Cloud Dienstes ist je nach Anwendungsfall von der Verarbeitung gigantischer Datenmengen abhängig. Die Kombination mehrerer Datenquellen kann dabei für transaktionale oder analytische Funktionalitäten in einem Produkt integriert werden. Bedingt durch die Analyse von dynamischen Datenquellen und der Verwaltung daraus gewonnener Informationen, steigt der Bedarf nach Betriebsressourcen und der Skalierbarkeit einer Anwendung. Beispielsweise erweitert sich der Datenbestand eines *Social Media* Portalanbieters durch Multimediadaten und Nachrichten der Anwender des Dienstes. Der Verwaltungsaufwand und die Analyse der erzeugten Daten ist in einem solchen Szenario von der Anzahl an Benutzern des Dienstes abhängig. Mit der durch die Verarbeitung riesieger Datenmengen gewonnenen Informationen, wie beispielsweise den *Text, Web* oder *Network data analysis* entspringenden Auswertungen, lassen sich neben ebenfalls ökonomisch geriebene Anzätze umsetzen [CML14]. Die Höhe des Datenaufkommens der letzten Jahre verringert sich aller Wahrscheinlichkeit nach auch nicht in der Zukunft [OY16].

Die Infrastruktur einer solchen Cloud-basierten Unternehmung beinhaltet verschiedene Ressourcen, die durch den Administrator des Dienstes zu orchestrieren sind. Der Dienst kann dabei auf eigener physischer Hardware oder auf der Infrastruktur eines Cloud Service Provider (CSP) betrieben werden. Die Nutzung und Verknüpfung verschiedener 'externer' *Cloud Services* ist dabei nicht ausgeschlossen und kann je nach Tarif zur Betriebskostensenkung der Anwendung beitragen. Basierend auf in *Marketplace*s angebotenen Diensten einer Vermittlungsschicht, sind die günstigsten Anbieter

einer bestimmten Cloudlösung identifizierbar. Diese *Cloud Broker* sind grundlegend mit der transparenten Vermittlung von Anfragen eines Cloud Service Custumer (CSC) an den passendsten Anbieter eines Dienstes verantwortlich [MG11]. Der Broker agiert demnach als zwischenliegende Schnittstelle von CSC und CSP. Die Vielfalt von auf eine Problemdömäne spezifizierten Brokerstrukturen unterstreicht den Komfort der Auslagerung von Beziehungsverhandlungen bei komplexen Aufgabenstellungen. Dabei können Abhängigkeiten zu CSPs in einem Broker auch transparent gehalten werden, was in einem eigenständigen Dienst resultiert, der dem Endanwender zur Verfügung gestellt wird [KJB15].Der Mehrwert eines Brokersystems kann ebenso auf nicht monetäre Eigenschaften einer Unternehmung bezogen sein und als zentrale Managementkomponente zur Steigerung der administrativen Effizienz bei der Umsetzung einer oder mehrerer Aufgaben eingesetzt werden. Ein sogenannter *Cloud Management Broker* zeichnet sich durch Einheitliche Schnittstellen zu CSPs, der Vereinigung von Legimitation der *Cloud-Subscriber* zu multiplen CSPs, sowie zu deren Programmierschnittstellen aus [nis17]. Die *NIST Cloud Computing Standards for Performance* verdeutlichen den Bedarf nach objektiver Einschätzung des Kosten- Nutzenverhältnisses eines cloudbasierten Dienstes. Darunter fallen unter anderem Leistungsmerkmale wie *Management, Benchmark* und *Service Lifecycle Performance* Aspekte, sowie deren Validierung durch Monitoring und Auditierung. Mit der Implementierung und Selbstevaluierung dieser Standards durch alle involvierten CSPs kann eine Brokerstruktur ihren Anwendern individuelle Performanzbedürfnisse zusichern [nis13]. Die von der jeweiligen Problemstellung abhängige Vermittlung von Diensten in einer multi-CSP Umgebung basiert in den meisten Fällen auf Implementierungen mehrerer, auf unterschiedlichen Technologien aufbauender Komponenten des Brokers.

Die Hauptproblematik, bei den für einen Anwendungsfall verwendeten Technologien, ist deren Stabilität, wohingegen die grundlegenden Anwendungskonzepte für eine generische Form der Verarbeitung wiederverwendbar sind [SV06]. Durch Konzepte des Model Driven Software Development (MDSD) kann die problemspezifische Arbeitsweise einer konkreten Technologie in Modellen abstrahiert werden. Mit dem Fokus auf das *was*, anstatt das *wie*, lassen sich verschiedene Problemstellungen einer Domäne zusammenfassen und generisch in Technologien übersetzen. Die Kombination von Konzepten mehrerer Domänen ist nicht ausgeschlossen und kann dem Anwendungsmodellierer ein mächtiges Werkzeug zur performanten Generierung von Anwendungscode komplex interagierender Technologien zur Verfügung stellen.

1.1. Problemstellung

Die Verarbeitung von komplexen Datenquellen benötigt neben Hardwareressourcen meist auch ein Ökosystem an softwarebasierten Werkzeugen. Die Zusammenfassung

dieser Softwaresammlung kann bei entsprechender Konfiguration Teil eines *Cloud Service* werden und sich der flexiblen Bearbeitung unterschiedlicher Datenquellen annehmen. Eine derartige Verarbeitungsplattform hängt immer zu einem gewissen Grad von konfigurierbaren Eigenschaften, wie beispielsweise einer Uniform Resource Locator (URL) oder XML Schema Definition (XSD) Struktur einer Datenquelle, ab. Um den Anwendern der mehrfach instanziierbaren und parallel betreibbaren Plattformen ein bequemes Anwendungsverhalten zu bieten, sind aus der Struktur einer Datenquelle ableitbare Eigenschaften in den jeweiligen Instanzen bereitzustellen. Neben der Verarbeitung von öffentlich zugänglichen Informationen kann eine solche Plattform demnach auch auf unternehmensinterne oder private Daten abgestimmt werden.

Beispielsweise verursacht die Arbeitsweise von neuen, sich in der Entwicklungsphase befindenden Technologien, in den meisten Fällen große Mengen an Logdaten und zusätzliche Informationen über angewandte Funktionalität. Die Ursache von auftretenden Unklarheiten und Fehlern in gesammelten Datensätzen kann jedoch nicht immer direkt bestimmt werden. Durch den Einsatz von Analysewerkzeugen oder gleichartigen Eigenentwicklungen, die sich auf die betreffende Fachdomäne beziehen, können Aktivitäten einer Anwendung nachvollziehbar dargestellt und mit zusätzlichen Informationen angereichert werden. Die Vorverarbeitung von Datensätzen trägt zur effizienten Bearbeitung identifizierter Kerninformationen bei und kann einem mehrschichtigen Modell folgen. Dabei sind neu gewonnene Erkenntnisse möglicherweise als Eingangsdaten für einen weiteren Verarbeitungsschritt mit einer spezifischen Plattform gekennzeichnet.

Die Verarbeitung von Daten einer bestimmten Domäne kann in manchen Situationen in einer abstrahierten Prozesskette verallgemeinert werden. Mit der Bereitstellung einer isolierten Verarbeitungsumgebung sind alle, für die jeweiligen Prozessschritte benötigte Anwendungen aufeinander abzustimmen. Softwareanwendungen lassen sich meist mit den vom Prozess abhängigen Attributen initialisieren, wobei durch deren Vorkonfiguration die *Usability* und Effizienz der Plattform gesteigert wird. Eine Brokerstruktur zur Verarbeitung von vorab definierten Datenquellen kann sich der Vermittlung von eigehenden Informationen an Anwendungen einer Plattform annehmen. Automatisch identifizierte Auffälligkeiten eines Datensatzes können, neben durch den Benutzer des Brokers manuell ausgeführten Anfragen, das auslösende Ereignis der Erstellung einer Plattform sein.

Handelt es sich bei involvierten Anwendungen um einen komplett konfigurierbaren und aufeinander abgestimmten Gesamtprozess, ist eine mit der Allokation von virtuellen Ressourcen vollautomatisch gestartete Plattform denkbar. Auf langfristige Sicht ist die Verwendung des Konzepts einer automatisierten Verarbeitungsplattform von Prozesse auslösenden Ereignissen abhängig. Die tatsächlichen Attribute eines Prozessauslösers stellen die grundlegen Verarbeitungsparameter der Plattform dar und können während der Instanziierung eines alle Anwendungen beinhaltenden *Cloud Image* be-

kannt gemacht werden. Die Initialisierung und Konfiguration von Applikationen ist in einigen Fällen, auf Grund fehlender Remote-Schnittstellen, nicht *on the fly* durchführbar. Sind involvierten Technologien von einer nicht automatisierbaren Nachkonfiguration abhängig, ist die Eigenentwicklung von Skripten bzw. Schnittstellen für die dynamische Ausführung von Systemfunktionalität nach der Instanziierungsphase ein denkbarer Lösungsansatz für derartige Plattformen. Somit sind wiederkehrende Verarbeitungsprozesse lediglich auf eine initiale Konfiguration angewiesen, die in Abhängigkeit des jeweiligen Auslösers automatisierbar ist. In einer solch semi-automatisierten Allokation sind beteiligte Aktoren bei auslösenden Situationen mit Informationen der assoziierten Plattform zu benachrichtigen. Beispielsweise haben in einer Unternehmung involvierten Abteilungen einer Organisation so die Möglichkeit, sich in voneinander abgeschirmten Eingabemasken innerhalb einer Prozessinstanz gegenseitig an zentraler Stelle zu Ergänzen.

Die dynamische Modifizierung der technischen Infrastruktur ist in einigen Fällen auch von Mitarbeitern durchzuführen. Das Abstrahieren von technischen Mechanismen und komplexen Abhängigkeiten zu einer durchdachten Zwischenschicht, soll auch technischen Laien den Umgang mit tiefgreifenden Aktionen innerhalb eines komplizierten Systems ermöglichen. Die Sicht des Anwenders auf Ressourcen bezogene Aktionen ist dabei in jedem Fall projektspezifisch, kann aber auf technologischer Ebene in Anwendungsmuster zusammengefasst werden.

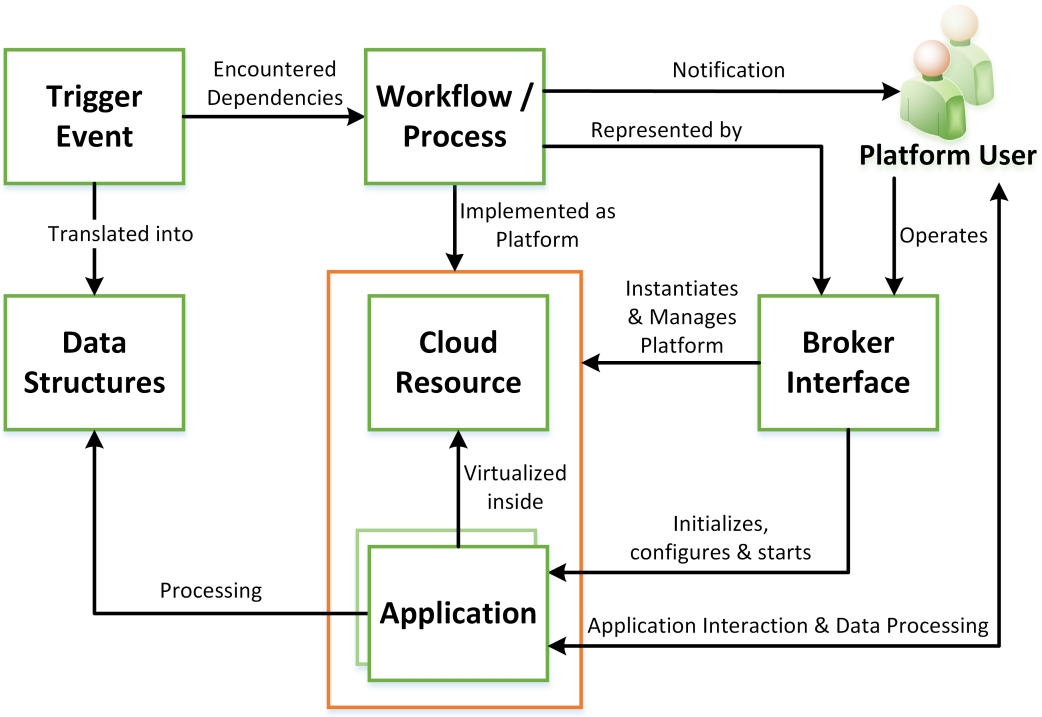

Abbildung 1.: Zusammenhänge der Relevanz von Plattform-Broker Infrastrukturen einer Organisation

Abbildung 1 verdeutlicht die bisher genannten Zusammenhänge einer problemspezifischen Cloudlösung im Bereich der Bereitstellung einer Plattform für die Datenverarbeitung von vorab definierten Quellmustern. Ausgehend von einem auftretenden Ereignis, sind Abhängigkeiten zu kausalen Teilprozessen, die letztendlich zu einem abschließenden Ergebnis führen, in einem Prozess bzw. *Workflow* zusammenzufassen. Eine darauf basierende Plattform ist auf die unterschiedlichen Anwendungen und Dienste einer Unternehmung angepasst und daher nicht dynamisch einsetzbar. Ebenfalls besteht die, den Prozessablauf verwaltende, *Broker* Komponente zu großen Teilen aus technologiespezifischen Eigenschaften, welche die Vermittlung von Ereignismustern einer Quelle in infrastrukturelle Anliegen übersetzt. Durch eine zentrale Schnittstelle sollen dem Plattform-User auch Möglichkeiten der manuellen Instanziierungen und Konfiguration von Plattformen zugänglich gemacht werden. Abhängig von der Bedienung einer Plattformspezifischen Software, können dem Endanwender Möglichkeiten der Interaktion mit Anwendungen einer Instanz in intuitiver Form, angeboten werden. Damit sollen auch Graphical User Interface (GUI)-basierte Endanwendungen innerhalb des *Broker*s durch *Remote-Desktop* Technologien zugänglich und von extern anwendbar gemacht werden.

1.2. Zielsetzung

Die vorliegende Arbeit beschäftigt sich mit problemspezifischen Cloudlösungen im Bereich der Bereitstellung einer Plattform für die Datenverarbeitung von vorab definierten Quellmustern. Innerhalb einer erweiterbaren Grammatik kann die Verbindung von Rohdaten und entsprechenden Verarbeitungsplattformen abgebildet werden. Das Hauptaugenmerk dieser Arbeit liegt auf der Erstellung einer generisch definierbaren und konfigurierbaren Architektur zur Vermittlung von vorab definierten Eingangsquellen und der Instanziierung von auf die Problemdomäne angepassten *Cloud Images*. Ein auf diese Domain Specific Language (DSL) abgestimmter Model to Code (M2C) *Compiler* ermöglicht die Generierung verschiedener Varianten von Modulen eines Anwendungsszenarios. Zur effizienten Formulierung neuer bzw. sich ändernder Eingangsquellen sind dafür verantwortliche Module an gekennzeichneten Stellen in der Struktur des generierten Quellcodes, mit individueller Logik anzureichern. Die automatisch generierte Code-Dokumentation beschreibt dabei auch Punkte, an denen der Programmierer bestimmte Zusatzfunktionalität, wie automatische Aktionen bei Auftreten eines bestimmten Attributes, nachtragen kann. Mit der Erstellung einer verarbeitenden Plattform erscheinen neue Abhängigkeiten, bezogen auf die Interoperabilität benötigter Anwendungen, sowie deren Konfiguration. Unter Verwendung der *OpenStack* Technologie kann, eine auf dem *Django*-Framework basierende *Broker*struktur, auf die Instanziierung von problemspezifischen Virtual Machine (VM)s und deren Vorkon-

figuration abgestimmt werden.

Hauptsächlich reduziert die DSL den Implementierungsaufwand generischer Problemstellungen bei der virtualisierten Verarbeitung von externen Datenquellen. Unterschiedliche Varianten komplexer Sachverhalte und mit Fremdsystemen verbundene Aktionen sind damit in wenigen Zeilen definierbar. Aus bekanntgegebenen Attributen wird die Zwischenschicht in Form eines *Dashboard*s zur Verwaltung virtualisierter Plattformen und Eingangsquellen bereitgestellt. Die Effizienz eines derart verschachtelten *Broker* soll in verschiedenen Anwendungsszenarien diskutiert werden. Dabei liegt das Hauptaugenmerk auf generischen Eigenschaften der DSL und dem Restaufwand während der individuellen Anreicherung der generierten Architektur mit entsprechender Anwendungslogik. Ebenfalls sind die mit der Erweiterung einer DSL verbundenen Auswirkungen auf M2C Definitionen und deren Implementierungen zu diskutieren.

1.3. Gliederung der Arbeit

Die in der vorliegenden Arbeit getroffenen Argumentationen eines generisch implementierbaren Plattform-Brokers basieren grundlegend auf den in Kapitel 2 beschriebenen Annahmen und Mechanismen. Der aktuelle Stand der Technik wird im Bezug auf Konzepte der Anwendungsvirtualisierung, Brokersysteme, sowie der Umsetzungen von Problemstellungen in der modellgetriebenen Softwareentwicklung in Kapitel 3 vorgestellt. Der in Kapitel 4 gegebene Überblick konkreter Technologien beschreibt, neben Rahemnwerken des **MGSE!** (**MGSE!**), auch mögliche Kandidaten für die Umsetzung eines spezifischen Brokers mit der generisch implementierbaren Architektur. Das Hauptaugenmerk dieser Arbeit besteht aus der in Kapitel 5 vorgestellten generischen Architektur für die Erstellung problemspezifischer, auf dynamische Eingangsdaten abgestimmter, Brokersysteme. Die in Kapitel 6 aufgeführten Anwendungsfälle sollen dabei für den vielfältigen Einsatz der vorgeschlagenen Architektur sensibilisieren. Die aus dieser Arbeit gewonnene Erkenntnisse werden in Kapitel 7 zusammengefasst und möglichen Erweiterungen gegenübergestellt.

2. Grundlagen

Die in der vorliegenden Arbeit angewandten Mechanismen basieren auf Grundlagen unterschiedlicher Themengebiete. Für ein tieferes Verständnis der Arbeit werden daher im Folgenden die wesentlichen Eigenschaften von Cloud Computing und insbesondere die Mechanismen von Cloud Brokersystemen vorgestellt. Ebenfalls werden die mit der modellgetriebenen Softwareentwicklung entstehenden Möglichkeiten diskutiert. Mit einem abschließenden Beispiel sollen elementare Annahmen über die Ziele dieser Arbeit verdeutlicht werden.

2.1. Cloud Computing

Das Konzept der Virtualisierung von Hardware erlaubt eine abstrakte und logische Sichtweise auf physische Ressourcen [BKNT11]. Diese fundamentale Herangehensweise an eine spezifische Problemlösung prägt die vorliegende Arbeit in nahezu jedem Kapitel.

Aus Hardware- und ökonomischer Sicht grenzt sich Cloud Computing nach [AFG+10] in drei Punkten von traditionellen netzwerkbasierten Infrastrukturen ab. Ein CSP stellt aus Benutzersicht unbegrenzte Ressourcen zu Verfügung, die durch *On-demand self-service* nicht im Vorhinein allokiert werden müssen. Die zeitliche Komponente der Instanziierung von Ressourcen verringert sich durch die Verwendung von Virtualisierungstechniken, was dem Endbenutzer im Umkehrschluss das Vorausplanen der Bereitstellung von Infrastrukturen abnimmt und spontane Kompositionen verschiedener Resourcen begünstigt. Aus CSP-sicht kann die der Virtualisierung zugrundeliegenden Hardware bei steigendem Bedarf erweitert werden. Bei der Einführung neuer Produkte bzw. Dienste eines Cloud Anbieters, können Rechenzentrum-interne Ressourcen mit zunehmendem Kundenbedarf ausgebaut werden. Die Kostenabrechnung bezieht sich auf die durch einen Dienst verbrauchten Ressourcen, aber nicht auf die z.B. theoretisch verfügbare Hardware.

Die Charakteristik des *Broad Network Access* nutzt standardisierte Mechanismen, um Ressourcen über das Netzwerk miteinander zu verbinden. Cloud Dienste nutzen die virtualisierte Infrastruktur eines CSP, deren physischer Zusammenhang durch Hypervisor Technologien abstrahiert ist und dem Nutzer durch *Resource Pooling* als einheitliches System dargestellt wird. Orte einer Ressource sind in diesem *multi-tenant* Modell transparent gehalten und werden bei Bedarf unterschiedlichen Konsumenten

zugeordnet. Die Skalierung einer Ressource kann in Abhängigkeit des momentanen Aufkommens an Verarbeitungsanfragen automatisiert werden, was als *Rapid Elasticity* bezeichnet wird. Im Gegensatz zu einer vertikalen Skalierung, bei der Ressourcen eines einzelnen Knotens aufgestockt werden, stellt die horizontale Skalierbarkeit eines verteilten Systems ist das Hinzufügen von weiteren Knoten dar. Prinzipiell ist der Verbrauch von Ressourcen in der Hypervisor Komponente messbar, um die Eigenschaft eines *Measured Service* verwirklichen zu können. Der von einem CSP angebotene Dienst steht dem CSC dabei nach Bedarf zur Verfügung (*On-demand self-service*). Dessen Abrechnung basiert jedoch, abhängig von dessen Geschäftsmodell, nicht auf dem erworbenen Dienst, sondern auf den damit verbrauchten Ressourcen [MG11]. Ein solcher *Measured Service* kann auch bedeutend zur Automatisierung einer Infrastruktur anhand definierter Regelsätze über die Orchestrierung virtualisierter Ressourcen in einem Cloud Management System (CMS) beitragen [Khe15]. Die Verwendung virtueller Ressourcen verschafft Unternehmen nicht nur Flexibilität im Hinblick auf kurzfristige Infrastrukturänderungen, sondern entlastet diese auch monetär durch die Einsparung von physischem Platz, Betriebskosten und Administrationsaufwand.

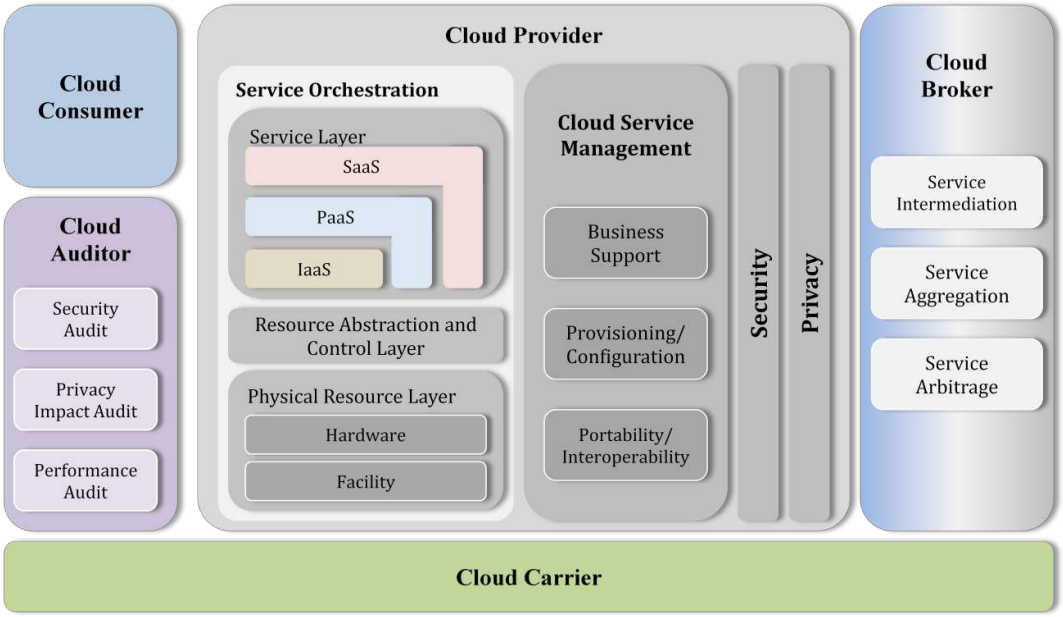

Abbildung 2.: Konzept des Cloud Computing Referenz Modell [LTM+12]

Die in Abbildung 2 dargestellte Refferenzarchitektur für Cloud Computing nach National Institute of Standards and Technology (NIST) identifiziert involvierte Aktoren, sowie deren Funktionalität und Kommunikation in diesem Ökosystem. Die Orchestrierung von Services eines CSP ist prinzipiell in drei aufeinander aufbauenden Schichten eingeteilt. Grundlegend wird ein Dienst durch die zugrundeliegenden, physischen, Hardware Ressourcen ermöglicht. Die in der mittleren Schicht angesiedelte Abstraktion von Ressourcen und deren Orchestrierung baut auf die physischen Ressourcen

auf und ist die Grundlage für *resource pooling, dynamic allocation* und *measured service*. Innerhalb des *Service Layer* definiert ein CSP Schnittstellen für die Konsumenten des angebotenen Dienstes. Die Infrastructure as a Service (IaaS) Schicht ermöglicht die grundlegende Virtualisierung von Infrastrukturen, prominente Beispiele sind *Amazon EC2* und *OpenStack*, mit Hilfe derer VMs bereitgestellt werden können. Darauf aufbauend können beispielsweise Platform as a Service (PaaS) im Stil von *Amazon Lambda* [VGO+16] betrieben werden, um dem Kunden eine vordefinierte Umgebung für eigens deklarierte Arbeitsschritte anzubieten. Ein CSP kann diese PaaS wiederum ausformulieren und als Applikation in Form eines Software as a Service (SaaS) zur Verfügung stellen [AK15, LTM+12]. Das Cloud Service Management eines CSP basiert auf der kundenseitigen Konfiguration eines Service und der Bereitstellung von Ressourcen durch den Anbieter. Getroffene Zusicherungen über die Performanz, Sicherheit und den Umgang mit privaten Daten eines CSC können durch die regelmäßige Auditierung eines Dienstes zur Herstellung einer Vertrauensbasis zu potentiellen Kunden beitragen.

Ein in der *Public Cloud* angebotener Dienst ist prinzipiell von beliebigen Individuen nutzbar. Typischerweise wird dem CSC in einem solchen Szenario der jeweilige *Cloud Service* in Rechnung gestellt [AGS15]. Mit dem Anbieten eines Dienstes im Internet ergeben sich zwangsläufig Bedenken im Bezug auf Limitierungen der Ressourcen eines CSPs, sowie der potentiellen Gefahr von *Vendor Lock-In* auf CSC-Seite [CPJ16]. Der Endanwender eines Cloud-basierten Dienstes ist nicht immer eine beliebige Entität. Das Prinzip einer unternehmens- oder organisationsinternen *Private Cloud* gewann in den letzten Jahren immer mehr an Bedeutung. Erschwingliche Hardware und professionelle Virtualiesierungssoftware ermöglichen Organisationen eine verschachtelte technische Infrastruktur, die hierarchisch auf einzelne Abteilungen abgebildet werden kann. Eine private Infrastruktur schränkt dabei den Zugriff von Diensten auf das Netzwerk des CSP ein [MG11]. Die virtuellen Betriebsmittel verschiedener Abteilungen einer Organisation können damit, trotz unterschiedlicher Fachdomänen, softwaregestützt in einer isolierten Infrastruktur miteinander interagieren.Mit der *Community Cloud* werden mehrere Anbieter einer bestimmten Problemdomäne zu einem gemeinsamen Dienst zusammengefasst, der von beteiligten Organisationen bzw. CSCs für die gemeinsame Umsetzung einer Problemstellung verwendet wird [MG11]. In einer solchen Föderation können ungenutzte Ressourcen in vorab ausgehandelter Art von Teilnehmern der Gemeinschaft verwendet und geteilt werden [CPJ16]. Die Kombination von den bisher genannten Cloudarten wird als *Hybrid Cloud* bezeichnet. Die darin involvierten CSPs sind separate Entitäten, die durch standardisierte Schnittstellen interagieren und von einem gemeinsamen Deployment an Ressourcen profitieren [AGS15].

2.2. Cloud Broker

Das Konzept eines Cloud Broker beruht auf der Vermittlung eines Dienstes oder der Aushandlung von Verträgen zweier Entitäten. Dessen grundlegende Verhaltensweisen sind Bestandteil der in Kapitel 5 beschrieben Architektur, wohingegen der folgende Abschnitt die konzeptionelle Funktionalitäten und die involvierten Aktoren der Vermittlungsschicht beschreibt.

Die Definition eines *Cloud Broker* ist nach NIST eine Entität, welche für das Management der Zustellung und Leistungsfähigkeit von Cloud Diensten verantwortlich ist [MG11]. Um die, über einen Dienst getroffenen Zusicherungen in einer flexiblen Umgebung wie der Cloud einzuhalten zu können, müssen CSPs nach [gar09] mit Cloud Brokern zusammenarbeiten. Diese Systeme sollen dem CSC Hürden im Bezug auf den Umgang mit multiplen CSPs und deren Dienste in Form einer zentralen und meist problemspezifischen Schnittstelle verringern. Die Überwachung der angebotenen Ressourcen und unterschiedlichen Technologien ist eine für den CSC interessante Eigenschaft bei der Auswahl des passendsten CSP. Aus Sicht eines CSP können diese Informationen ebenfalls zur Steigerung des Verständnisses für Marktverhältnisse führen und einer Erstellung neuer Vermittlungsstrukturen mit potentiellem Mehrwert verwendet werden. Der Mehrwert dieser Vermittlungsschicht ergibt sich aus der Entkopplung beider Aktoren und den Managementdiensten, die von mehreren Anbietern abhängig sind und eine dementsprechend komplexe Aushandlung von Schnittstellen erfordern [CPJ16].

Wie in Abbildung 3 dargestellt, besteht ein direktes Verhältnis zwischen dem CSC und einer Brokerstruktur bei der Anwendung des Vermittlungsdienstes. Der Broker ist damit aus Kundensicht ebenfalls ein CSP. Die innerhalb eines Brokerdienstes involvierten Angebote von CSPs bauen ein indirektes Verhältnis zu den CSCs auf, welcher damit auch indirekt den jeweiligen Geschäftsbedingungen zustimmt. Ist beispielweise der Umgang mit Informationen innerhalb der aufeinander aufbauenden Dienstketten an einer beliebigen Stelle mit Einschränkungen der Datensicherheit verbunden, muss diese Eigenschaft durch die Vermittlungsschicht, zur Wahrung der Vertrauensbasis mit den CSCs, kenntlich gemacht werden.

Der durch den Broker angebotene Dienst kann prinzipiell in drei Kategorien unterteilt werden:

- *Service Intermediation*: Die Angebote der CSPs werden durch diese Zwischenschicht mit einem zusätzlichen Mehrwert erweitert. Beispielsweise erleichtert eine gemeinsame Verwaltung des Zugriffs auf bereitgestellte Dienste dem Anwender eines Brokers die Auswahl eines passenden CSP.

- *Service Aggregation*: Neben gesteigerter Flexibilität von Infrastrukturen ermöglicht der Betrieb von IaaS, PaaS und SaaS Lösungen neben den in Abschnitt 2.1

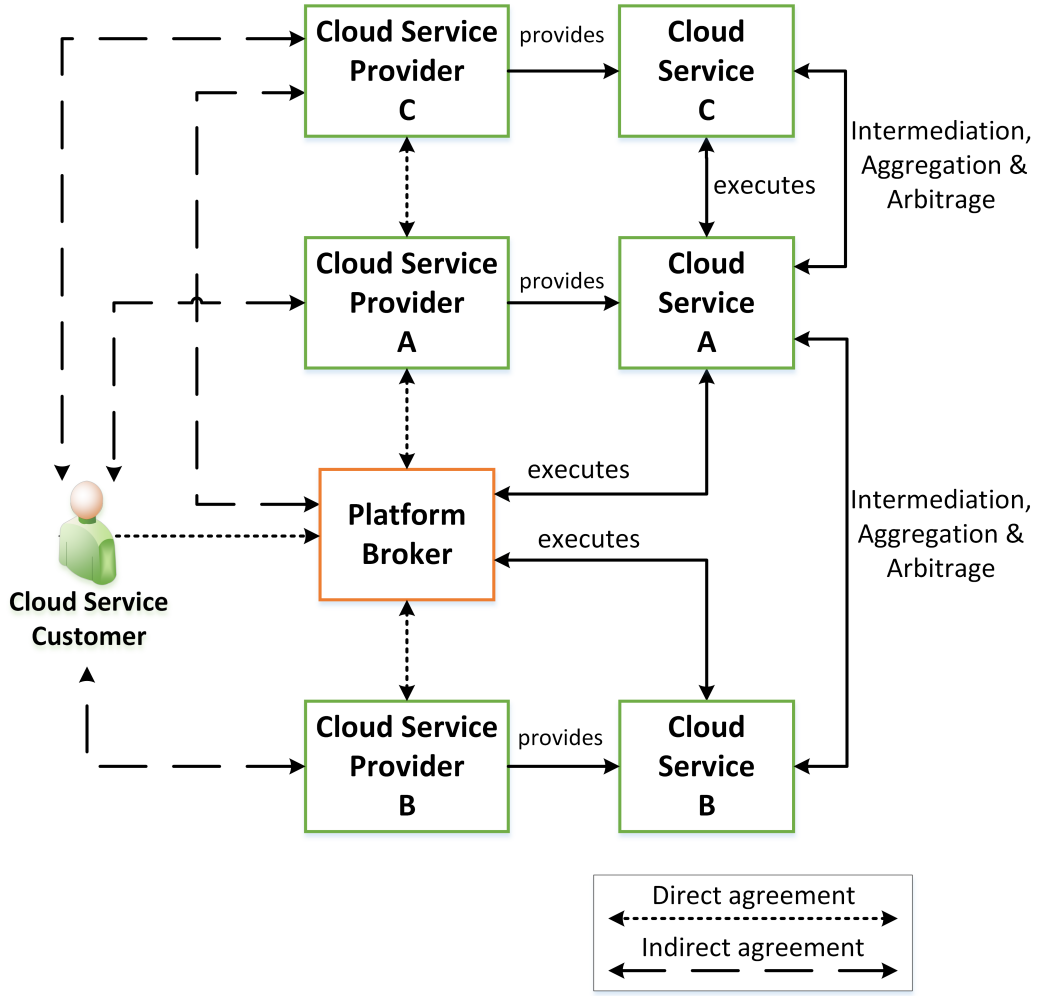

Abbildung 3.: Konzept eines Cloud Brokers - Eigene Darstellung in Anlehnung an [LTM+12]

beschriebenen Ansätzen auch Geschäftsmodelle durch Aggregation bestehender Dienste eines Drittanbieters. Durch die Kombination und Integration mehrerer Dienste unterschiedlicher CSPs in einem Angebot lassen sich ebenfall neue *Cloud Service*s erstellen. Der Broker ist dabei für die Einhaltung zugesicherter Eigenschaften des Dienstes verantwortlich und daher von allen involvierten CSPs abhängig.

- *Service Arbitage*: Stehen einem Broker mehrere CSPs mit gleichartigen Diensten zur Verfügung, kann einem Kunden die Auswahl über den Grad der Aggregation von Diensten oder die simple Auswahl von Angeboten im Stil eines *Marketplace*, bereitgestellt werden. Die Flexibilität bei der Auswahl eines Angebots kann auch automatisiert und für den Kunden transparent gehalten sein. Die Messung von Attributen verfügbarer Dienste, wie deren Antwortzeit und durch den Benutzer an den Broker übermittelte Anforderungen, wird mit dem angemessensten Angebot beantwortet.

Die Einhaltung der in den Service Level Agreement (SLA)s definierten Eigenschaften eines Dienstes, wie beispielsweise der regelmäßigen Auditierung betroffener Ressourcen, ist bei interoperablen bzw. aggregierten Infrastrukturen unterschiedlicher CSP durch Cloud Broker zu realisieren. Bei konkurrierenden CSP und der Anwendung von *Broker* Technologien, kann dem Kunden eines Cloud Service ebenfalls die Wahl der Laufzeitumgebung freigestellt werden [LCX12]. Unterschiedliche funktionale und nicht funktionale von CSP beworbene Eigenschaften eines Dienstes sind auch grundlegend für *Service Broker* mit eigens verwalteten *Marketplace* Strukturen [SBBS12]. Für die Orchestrierung erworbener Ressourcen wird dem Endnutzer in der Regel eine Schnittstelle zum System bereitgestellt, in der technischen Details transparent gehalten sind. Diese Art der Selbstadministration von dynamischen Ressourcen durch den CSC ist auf die entsprechende Zielgruppe zugeschnitten und beinhaltet meist nur ausgewählte Aktionen über Betriebsmittel eines erworbenen Dienstes [AGS15].

Die Aggregation unterschiedlicher Anwendungen innerhalb eines Dienstes lässt sich durch auftretende Interaktionsmuster von klassischen Geschäftsprozessen und Prozessketten ableiten [BKNT11]. Eine weitere Übersetzung dieses Ablaufs in abgestimmte Aufrufe von Funktionalitäten der agierenden Anwendungen, ermöglicht die Bereitstellung einer Plattform für die Ausführung von spezifizierten Unternehmungen. Besitzt dieses Ökosystem von Anwendungen ebenfalls eine zentrale Schnittstelle für die Bedienung der gesamten Plattform, können derartig voneinander abhängige Prozesse innerhalb eines *Cloud Image* zusammengefasst werden. Die Instanziierung dieses *Image* resultiert in einer vollwertigen VM, die über eine Internet Protocol (IP) Adresse ansprechbar und unter einem spezifizierten Port zu bedienen ist [KFF+08].

Die vorliegende Arbeit grenzt sich zu traditionellen Broker Systemen ab, die zwischen CSC und Diensten eines CSP vermitteln. Vielmehr wird in Kapitel 5 auf eingehende Informationen des Brokers, die zur Konfiguration einer Verarbeitungsplattform verwendbar sind und dem CSC eine Auswahl an zu bewältigenden Problemstellungen anbietet, eingegangen. Der Informationsrückfluss von instanziierten Plattformen ist jedoch eine Gemeinsamkeit mit den *Monitoring*-Eigenschaften der Vermittlungsschichten.

2.3. Modellgetriebene Softwareentwicklung

Abstraktion ist eine traditionelle und mächtige Vorgehensweise, um Anwendungsentwickler von der Komplexität abzuschirmen [Jö13]. In der klassischen Softwareentwicklung basiert der geschriebene Code meist auf Rahmenwerken und Technologien, deren Funktionalität durch programmatische Aufrufe bereitgestellter Schnittstellen, für die Umsetzung individueller Logik verwendet wird. Mit der Implementierung eines Softwaresystems verringert sich auf Grund problemspezifischer Anforderungen und der

Wahl einer bestimmten Architektur in den meisten Fällen die Wiederverwendbarkeit von Modulen. Die individuelle Ausformulierung von Abhängigkeiten einzelner Komponenten eines komplexen Systems birgt neben der langwierigen Begutachtung von technologiespezifischer Kommunikation an jeder auftretenden Stelle im Code, auch die Gefahr von Qualitätseinbußen. Sind mehrere Entwickler in ein Projekt involviert, ergeben sich aus unterschiedlichen Programmierstilen verschieden benannte Attribute und Funktionen, sowie qualitative Differenzen bei deren Dokumentation. Durch Modularisierung von Anwendungslogik kann eine wiederverwendbare Architektur erschaffen werden. Mit deren Veraltung auf Grund sich ändernder Technologien, sind alle betroffenen Module zu überarbeiten und die daraus resultierende Anwendungslogik mit den restlichen Komponenten zu verknüpfen. Bei der Definition neuer Anforderungen oder Funktionalitäten müssen diese, möglicherweise der Architektur wiedersprechenden, Programmteile ausformuliert und in das System eingepflegt werden.

Die Verwendung von Modellen bietet auf Grund von wiederverwendbaren Artefakten, die eine spätere Implementierung repräsentieren und keine Anwendungslogik beinhalten, eine höhere Ebene der Abstraktion von domänenspezifischem Wissen. Diese, durch eine DSL beschreibbaren Strukturen und Interaktionen, sind für einen Domänenexperten leichter zu verstehen wie deren letztendliche Darstellung in Form von Programmcode [SV06, VBD+13]. Ziele der MDSD sind neben verringerter Entwicklungszeit auch gesteigerte SW Qualität, Separation of Concerns, Wiederverwendung und Interoperabilität bzw. Portabilität [Jö13, SV06, VBD+13].

Im Gegensatz zu General Purpose Language (GPL)s, wie C++, Java oder Python, ermöglicht die DSL einen deklarativen Programmieransatz, mit dem anstatt der konkreten Umsetzung des Anwendungsmodells dessen funktionale Begebenheit ausgedrückt wird [AC13, VBD+13]. Diese technologieunabhängige Formulierung besteht in den meisten Fällen aus nicht Turing-vollständigen Strukturen, mit denen nur eingeschränkte Abstraktionen durch Code bestimmbar sind. Deren Einsatz wird durch verschiedene Einflussfaktoren bestimmt und rechtfertigt sich abhängig vom Kontext. Die Größe einer von wenigen Entwicklern und Domänenexperten erstellte DSL ist durch den Bezug von Syntax und Semantik zu einem bestimmten Aufgabenbereich immer überschaulicher, als die von allgemeingültigen GPLs[VBD+13].

Mit der Anwendung von Transformationstechniken auf eine DSL, lassen sich technologiespezifische Codefragmente und deren Relationen zur benutzenden Anwendung erstellen. Die dafür verantwortliche *Template Engine* interpretiert einzelne Artefakte des Modells und ist für das Generieren von Architekturinstanzen in unterschiedlichen Programmiersprachen, welche in konkreten *Template*s ausformuliert sind, verantwortlich. Mit der Abbildung allgemeingültiger Modelle und Konzepten von Technologien ergeben sich verschiedene Vorteile im Bezug auf ein dynamisches Anwendungsszenario.

Alle unveränderlichen domänenspezifischen Eigenschaften sind in Konzepte zusammenzufassen und durch Strukturen der jeweiligen Zielplattform anzuwenden. Auf die DSL angepassten Code Generatoren verbinden die Syntax und Semantik der Sprache meist mit der Anwendung von Logik konkreter Technologien oder Konstruktion von Modulen in Hochsprachen wie beispielsweise Ruby oder Python. Die auf dieser ausführenden Ebene basierende *Execution Engine* kann jedoch auch durch eine kompilierte Laufzeitumgebung und direkten Interpretation und Anwendung von deklarierten Strukturen der DSL Instanz beruhen. Jedes Programm mit Bezug zu einer bestimmten Domäne besteht zu einem gewissen Teil aus individueller Logik. Um dieser Variabilität gerecht zu werden, muss die DSL prägnante Abstraktionen für die Definition von Grundstrukturen einer verallgemeinerten Ausprägung des jeweiligen Features, bereitstellen.

Bei der Erstellung einer Menge von gleichartigen Anwendungssystemen ist der einmalige Definitionsaufwand einer DSL ein verkraftbarer und fruchtbarer Mehraufwand. Die Mächtigkeit des MDSD Vorgehens bestärkt das *separation of concerns* Prinzip, ist jedoch auf die Zusammenarbeit mehrerer Rollen, welche das jeweilige Fachwissen bereitstellen, angewiesen. Abbildung 4 veranschaulicht den Ablauf der Erstellung und Anwendung einer DSL und zeigt Aktoren, welche in den Prozess involviert sind.

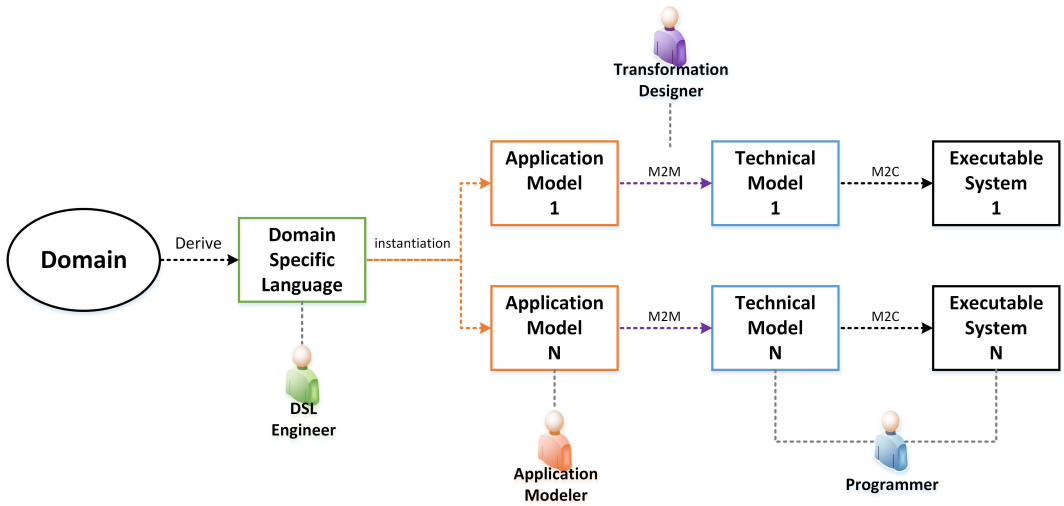

Abbildung 4.: Rollen in MGSE - Eigene Darstellung in Anlehnung an [Jö13, SV06]

Die von einer Problemdomäne abgeleitete DSL definiert ein Metamodell und beinhaltet eine konkrete Darstellung, die sich auf ein spezifisches Meta-Metamodell bezieht. Dabei entscheidet der *DSL Engineer* über die Abbildung der Komplexität und Tiefe von domänenspezifischen Konstruktionen und Mechanismen innerhalb der eigenen Modellierung. Die eigentliche Domäne kann neben verallgemeinernden und weitgefächerten Problemen wie "Embedded Systems" oder "Bank System" selbstverständlich auch als Abstrahierung auf Technologieebene wie "Apache Storm" oder

"Spark" identifiziert werden. Semantisch ist eine DSL auf die Konzeptionellen Begebenheiten involvierter Problembereiche abzustimmen und in Form von Regelsätzen zu deklarieren.

Das damit formulierte Metamodell beinhaltet *Constraints*, welche mittels textueller Grammatik oder der Festlegung möglicher Bausteine, Annotationen und Relationen in grafischen Modellierungsumgebungen ausgedrückt werden kann. Das Modell einer konkreten Anwendung repräsentiert die Struktur eines domänenspezifischen Problems und wird durch einen *Application Modeler* instanziiert. Die Übersetzung eines Anwendungsmodells in unterschiedliche technische Darstellung ermöglicht den Einsatz verschiedener Ansätze der Problemlösung bei gleichbleibender Allgemeingültigkeit von Systembeschreibungen und Anforderungen. Die durch den *Transformation Designer* erstellten *Template*s beinhalten die DSL abbildende Funktionalität und stützen sich dabei meist auf eine spezifische Technologie. Das für die jeweilige Vorlage benötigte Fachwissen ist unter Zusammenarbeit mit einem *Programmer* zu bestimmen und einzupflegen. Für die Transformation von Modellen in ausführbare Systeme sind in der Mehrheit der Fälle verschiedenste Module mit individueller Anwendungslogik anzureichern. Die aus der M2C Transformation resultierende Anwendung ist eine konkrete Ausprägung der durch den *Application Modeler* definierten Problemlösung.

Das Generieren von ausführbarem Code aus einem Modell verringert in erster Linie die Entwicklungszeit eines domänenspezifischen Systems. Der Gebrauch von *Parser*n ist dabei unablässlich, da Strukturen validiert und für die Transformationsschritte verwendet werden. Der Einsatz von MDSD Techniken rechtfertigt sich damit, dass Konzepte stabiler wie Technologien sind [SV06].

In der klassischen Softwareentwicklung basiert der geschriebene Code, sowie darauf bezogene Unit Tests, auf dem Verständnis über Anforderungen. Bei der Validierung von Instanzen einer DSL liegt der Fokus auf Anforderungen an das System. Einzelne Verzweigungen innerhalb der generierten Software sind entweder durch den Code-Generator zu testen, manuell zu implementieren oder zusammen mit dem Produktivcode zu generieren[VBD+13]. Beispielsweise kann innerhalb des *Xpand* Code Generator eine Menge an Tests auf Transformationen angewendet werden[VBD+13]. Modelle sind abstrakt und gleichzeitig Formal [SV06]

Die Erstellung von Softwaremodellen dient nicht alleine der Dokumentation einer Architektur, sondern kann auch für die Generierung von ausführbarem Quellcode verwendet werden.

Jede implementierte DSL ist individuell und durch die Teildomäne getrieben, auf die sie sich bezieht. Dennoch lassen sich viele gemeinsame Paradigmen bei der Beschreibung einer DSL finden. Oft bieten Meta-Metamodelle eine logische Struktur wie *Namespaces* oder Module für die Instanziierung innerhalb einer DSL an. Die Sichtbarkeit einzelner Module bezieht sich auf in Strukturen des Metamodells zu-

sammengestellten Abstraktionen der Ausgangsdomäne. Partitionierungen von generierter Anwendungslogik in physisch getrennte Einheiten, wie beispielsweise Dateien, sind nicht zwangsläufig durch Artefakte der DSL widergespiegelt. Das Ändern oder Festlegen der Organisation und Verschachtelung einzelner Komponenten, hat direkte Auswirkungen auf die Logik aller davon abgeleiteten Modelle. Bei ungeschickter Bestimmung der Partitionierung können selbst kleinste Änderungen, wie die Benennung von Modulen oder Regeln, die kausale Überarbeitung aller unterliegenden Schichten bedeuten.

Die geschriebene Sprache hat das Ausmaß von Zeit und Raum der Kommunikation der Menschheit grundlegend verändert. Eine Konversation in heterogenen Sprachkonstrukten benötigt eine vermittelnde Komponete, was ebenfalls für die Transformation von Programmiersprachen und Modellen gilt [PYS11]. Abbildung 5 zeigt den Proze-

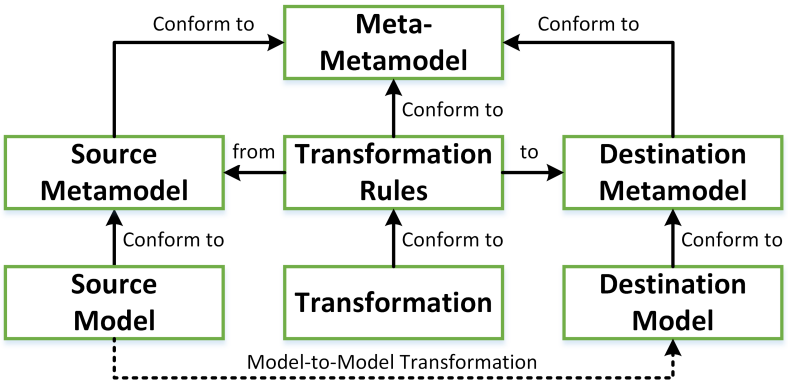

Abbildung 5.: Prozess von Model-to-Model Transformationen - Eigene Darstellung in Anlehnung an [LA14]

saufbau einer Model to Model (M2M)-Transformation. Eine Transformationsregel findet im ersten Schritt dieses Prozess die Abbildung von korrespondierenden Konzepten beider Technologien. Gemeinsame Strukturen der in einer Transformation involvierten Metamodelle oder aus den Strukturen modellierbare Konzepte sind die Grundlage für eine erfolgreiche Übersetzung. Alle Instanzen des Quell-Metamodells sind bei der Anwendung einer validen Transformation in Instanzen des Ziel-Metamodells abbildbar. Eine sogenannte *Transformation Engine* oder *Execution* ist dabei mit der automatischen Generierung des Zielmodells in Abhängigkeit des Quellmodells beauftragt [LA14].

Formale Sprachen sind die Basis der meisten Aktivitäten in der Informatik. Mit gesteigerter Verwendung von Konzepten des MDSD wurden Modellierungsumgebungen populär, in denen die Erstellung von Grundgerüsten einer Menge an gleichartigen Anwendungen mittels grafischer Werkzeuge ermöglicht wurde. Eine Grafische Syntax besitzt im Gegensatz zu textuellen Darstellungen keine vordefinierte Reihenfolge bei deren Betrachtung, stellt jedoch auftretende Relationen und Quantitäten einzelner

Module anschaulich dar [HJK+09].

Am Beispiel der Formulierung einer Zustandsänderung wird im Folgenden der Unterschied zwischen generischer und *custom* Syntax verdeutlicht. Die Erstellung einer Syntax geschieht auf der Ebene der Metamodellierungssprache und beinhaltet eine konkrete Struktur, mit der alle für eine Transition benötigten Attribute deklarierbar gemacht werden. Mit der Anwendung einer Regel in der Form von `Transition {source : State "StateA" target: State "StateB"}` sind mehrere Schlüsselwörter und darauf bezogene Zeichenketten durch einen Anwendungsmodellierer festzulegen. Dieser langatmige Satzbau ist ein allgemeingültiger Ausdruck, der nicht auf die Semantik der domänenspezifischen Sprache eingeht und nach Belieben mit weiteren Regelsätzen erweiterbar ist. In den meisten Fällen können die Schlüsselworte und Attribute in eine semantische Beziehung zur jeweiligen Funktionalität der Regel gesetzt und durch Symbolik wie `StateA` $\rightarrow$ `StateB` dargestellt werden. Dabei wird der erste Parameter als `source` behandelt und durch ein symbolische Veranschaulichung in den nachfolgenden `target` Zustand übersetzt. Das $\rightarrow$ Symbol ist in diesem Fall das in einem *Compiler* verwendete Schlüsselwort für die Generierung einer Transition im Anwendungscode. Prinzipiell kann aus beiden Formen der Syntax identischer Anwendungscode generiert werden. Eigens definierte Symbolik und Regelstrukturen sollten in jedem Fall ersichtliche Funktionalität beschreiben [HJK+09].

Der Model Driven Architecture (MDA) Ansatz basiert auf drei grundlegenden Konzepten. Mit dem Computation Independent Model (CIM) sollen benutzerspezifische Anforderungen der DSL repräsentiert werden, die nicht von Bedeutung für die funktionale Verarbeitung sind. Mit einem darauf basierenden Platform Independent Model (PIM) werden die Anwendungen unabhängig von ihrer Laufzeitplattform modelliert. Aus einem solchen Anwendungsmodell kann letztendlich ein Platform Specific-Model (PSM) generiert werden, das auf eine spezifische Plattform bzw. Technologie angepasst wurde [LBR16].

2.4. Bedarf nach Anwendungsspezifischen Infrastrukturen

Die Infrastruktur einer Unternehmung wird grundsätzlich durch zu erfüllende Aktionen und Dienstleistungen bestimmt. Abteilungsinterne *Private Clouds* ermöglichen eine dynamische Entwicklungsumgebung für projektspezifische Problemstellungen bei gelichzeitiger *Separation of Concerns* mit Produktivumgebungen oder unautorisiertem Personal.

Der folgende Abschnitt behandelt grundlegende Gedanken zur Anwendung von Virtualisierungstechnologien in Verbindung mit der Verarbeitung von externen Datensätzen. Die jeweiligen Szenarien repräsentieren Teilaspekte der in Kapitel 5 beschriebenen Architektur, beziehen sich aber auf die grundlegende Motivation der Erstellung eines

mit der Verarbeitung von Datenquellen betrauten *Broker*.

Entwicklungsphasen

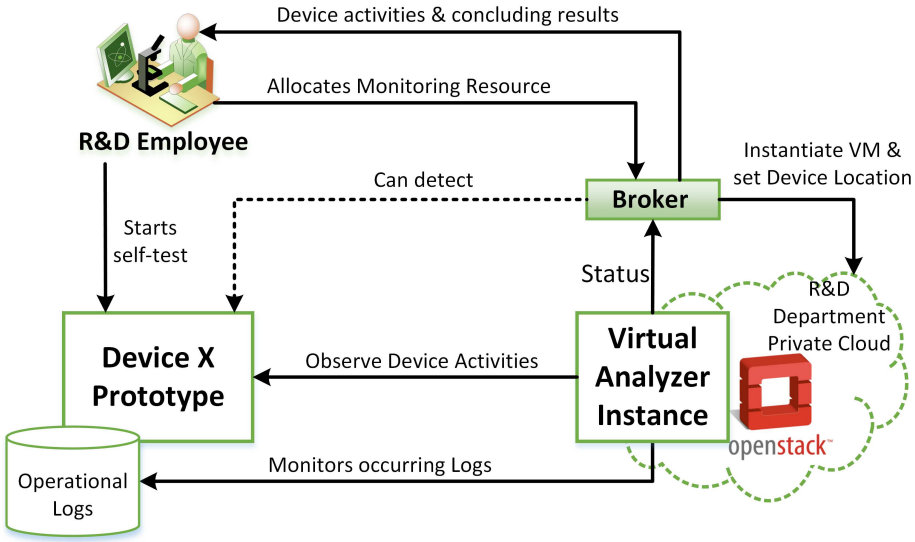

Abbildung 6.: Infrastrukturbedarf am Beispiel von Research and Development

Wie in Abbildung 6 am Beispiel des Selbsttests eines Prototypen aufgezeigt, ermöglicht die Verwendung von Virtualisierungstechnologien einen gewissen Grad an Flexibilität bei der Umsetzung von automatisierten Analyseinfrastrukturen. Durch eine abteilungsinterne *Private Cloud* werden Instanzen einer VM nach Bedarf von einem *Image* gestartet und anschließend mit der Analyse von operationalen Logdaten eines spezifizierten Prototyps betraut. Für einen performanten und bequemen *Workflow* beinhaltet das *Image* eine auf die aufkommenden Logdaten abgestimmte Analysesoftware, die Verknüpfung mit der Hardware des Prototypen findet während der Instanziierung der VM statt.

Als Grundlage für dieses Vorgehen kann eine semi-automatische, auf die Problemstellung abgestimmte Vermittlungsschicht implementiert und den Entwicklern zur Verfügung gestellt werden. Dabei hat diese Broker-ähnliche Struktur Kenntnis über Arbeitsschritte zur Allokation von virtuellen Ressourcen innerhalb des Unternehmens, sowie einen Mechanismus zur Detektion des zu betrachteten Subjekts/Prototyps. Für die Steigerung der Quality of Experience (QoE) des Anwenders dieser Zwischenschicht, können Informationen über den momentane Zustand des Prototyp tabellarisch aufbereitet und mit einer direkten Aktion, z.B. der Allokation einer Ressource, angereichert werden. Ausgeführte Aktionen mit Auswirkungen auf die Infrastruktur der Organisation sind nicht zuletzt aus Gründen der Netzwerkdokumentation innerhalb des *Content Management System* festzuhalten, viel mehr führt eine Rückkopplung von resultierenden Informationen an den Broker zur Verdichtung der Problemdomäne. Bereits

instanziierte VMs antworten dem Broker, der einzelne Attribute der rückfließenden Analyeergebnisse visualisiert und eine Verbindung auf die jeweilige Maschine zu einer genaueren, manuellen, Betrachtung anbietet. Diese mit einem *Dashboard* vergleichbare Darstellung von Informationen kann ebenfalls auf die Koexistenz unterschiedlicher Varianten des Prototyps eines Produkts abgestimmt sein.

Auffällige Ergebnisse einer bestimmten Analyse sind an zentraler Stelle identifizierbar, das Bereitstellen der VM an Mitarbeiter geschieht durch das Vergeben von Rechten über die Ressource. Global stationierten Unternehmen wird so eine bequeme Möglichkeit von beispielsweise *Remote Reviews* der Analyseergebnisse gegeben. Durch *Acountability*-Eigenschaften von CMS Technologien können mögliche Aktionen eines Anwenders in dieser Schicht festgehalten und weiter eingegrenzt werden.

Analyse und Verarbeitung unternehmensexterner Daten

Der Betrieb von abteilungsinternen Hypervisortechnologien kann ebenfalls mit maßgeschneiderten Abläufen und den dadurch reduzierten menschlichen Fehlern, sowie gesteigerter Performanz, argumentiert werden. Grundlegend ist die Verknüpfung einer Anwendung mit jeder zugänglichen Datenquelle denkbar. Bei der Kommunikation mit Ressourcen außerhalb des Firmennetzwerks ist die *Private Cloud* dementsprechend zu konfigurieren.

Die Struktur einer Datenquelle zeichnet sich insbesondere durch auftretende Metadaten aus, welche sich als erste Grundlage für das Design von aufgabenbezogenen Infrastrukturen eignen[1]. Hängen Datensätze mehrerer Domänen mit einem Anwendungsszenario zusammen, sind Gemeinsamkeiten zu identifizieren und in einer hierarchischen Struktur, welche den Anwendungsfall repräsentiert, für die spätere Darstellung der Brokeraktivitäten in einer GUI zu berücksichtigen. In komplexen und zusammenhängenden Datensätzen, können Unterschiede einzelner Attribute von Metadaten eine bestimmte Bedeutung haben und automatisiert erkannt sowie in eine direkte Reaktion übersetzt werden.

Der Einsatz von *Stream Processing* Technologien rechtfertigt sich nicht zuletzt mit dem Aufkommen von unterschiedlichsten Datenquellen der modernen IT. Vielmehr ermöglicht Paralleles Complex Event Processing (CEP) nahezu die Echtzeitberechnung von Datenströmen, welche zwischen unabhängigen Instanzen aufgeteilt und darin verarbeitet werden. Das frühe Erkennen von Trends eines Datenstroms ist ein vielseitiger Grundgedanke und wird bereits im Rahmen der Geschäftsanalyse ein-

[1] Metadaten stellen beschreibende Informationen über einen Datensatz dar. Beispielsweise sind Sender und Empfänger einer Email, deren Sendezeit und Betreff als Metadaten aufzufassen. Die tatsächliche Nachricht wird durch diese Meta-Informationen beschrieben. Die Verwendung derartiger Strukturen kann je nach Anwendungsfall und auftretenden Attributen zu einem aussagekräftigen Gesamtbild verarbeitet werden.

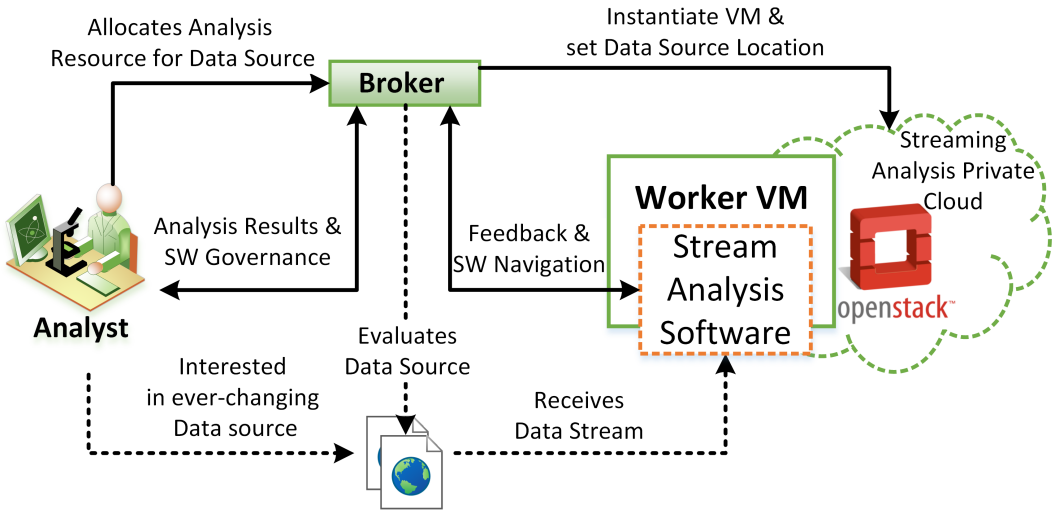

Abbildung 7.: Trennung der Schnittstelle in Anwender-Sicht und Verwendung von Cloud Technologien

gesetzt [MMTR16]. Wie in Abbildung 7 dargestellt, ist die *Broker* Komponente mit der Begutachtung von zu analysierenden Datenquellen, sowie möglicherweise der Einhaltung existierender Restriktionen für eine abteilungsinterne Kommunikation (oder Blacklists) betraut. An dieser Stelle sind ebenfalls mehrere automatisierte Umsetzungen zur Performancesteigerung, wie beispielsweise die Übersetzung der entsprungenen Metadaten einer Datenquelle in XSD Notation oder der Suche nach passenden Templates für die Analysesoftware in Repositorien des Softwareanbieters, möglich. Mit der Analyse des *traffic*s der Quelle ist der *Broker* bei entsprechender Implementierung in der Lage, eine erste Einschätzung über den Bedarf der für die Analyse benötigten Ressourcen zu treffen. Verschiedene Instanzen einer Technologie können gemeinsam instanziiert und aufeinander abgestimmt werden, bevor die tatsächliche Verarbeitung der Datenquelle automatisiert in einer verteilten Umgebung startet. Hierbei ist die zentrale Steuerung der verteilten Plattform zu berücksichtigen.

Relevanz der zu analysierenden Daten

Wie in jeweils Abbildung 6 und 7 aufgezeigt, separiert der *Broker* den Anwender eines Dienstes von technischen Aspekten des Hypervisors oder CMS. Mit dieser unterstützenden Logik kommen jedoch auch Gefahren im Bezug auf die Zweckentfremdung oder Verschwendung von Ressourcen. Schon bei der Auflistung von Eingangsdaten des Brokers kann dies durch Plausibilitätsüberprüfungen vorgebeugt werden. Je nach Anwendungsfall ist eine Vorverarbeitung der Daten durch ein separates System angebracht, indem eine Aufbereitung der Datenstrukturen in ein auf den Broker angepasstes Konzept geschieht. Dieser Punkt wird in Abschnitt 5 wieder aufgegriffen und soll den Einsatz von Brokern als einzelnen Schritt einer längeren Verarbeitungskette rechtfertigen.

3. Verwandte Arbeiten

Im Folgenden wird für die vorliegenden Arbeit relevante Literatur im Bezug auf Konzepte der Virtualisierung von Anwendungen, den unterschiedlichen Einsatzgebieten und Verhaltensweisen von Brokersystemen, sowie die Umsetzungen von Problemstellungen auf Grundlage der modellgetriebenen Softwareentwicklung, vorgestellt. Neben den technischen Begebenheiten einzelner Architekturen der referenzierten Artikel, soll dieses Kapitel auch für die Vielfältigkeit bestehender Systeme sensibilisieren. Die in diesem Kapitel behandelten Prinzipien und Muster helfen bei der Einordnung der vorliegenden Arbeit, sowie der Abgrenzung von bereits bestehenden Systemen.

3.1. Virtualisierung von Anwendungen und Cloud Technologien

Das Prinzip der Virtualisierung ist ein in der Informationstechnischen Landschaft weit verbreitetes Paradigma. Neben einem starken Bezug zu Sicherheitsaspekten ist die Bereitstellung virtualisierter Ressourcen eine mittlerweile gängige Herangehensweise an die Umsetzung dynamischer Dienste. Mit der Virtualisierung von Netzwerkressourcen ergeben sich neue Möglichkeiten der Verknüpfung und Administration von Diensten. Durch derartige Technologien können je nach Anwendungsfall ökonomische Einsparungen ermöglicht werden.

Konzepte der Virtualisierung von Desktops

Das 2004 in [BPSN04] vorgestellte *MobiDesk* System für mobiles virtualisiertes Desktop Computing entkoppelt Sitzungen von Maschinen eines Endanwenders von der Hardware eines Anbieters, auf der die letztendliche Anwendungslogik bereitgestellt wird. Die Clients eines solchen Systems sind zustandslose Anwendungen, die Benutzereingaben entgegennehmen, an die entsprechende Maschine eines Anbieters weiterleitet und deren Antwort visualisieren. Der mobile Aspekt dieses Systems bezieht sich auf die Migration von Sitzungen einzelner Benutzer auf andere Hosts zur Laufzeit. Das System besteht aus einer Menge an Backend Session Servern, Komponenten zur Datenhaltung und mehreren Client Anwendungen, die mit den Proxy Servern zum Backend kommunizieren. Mit Einführung eines *Thinv Virtualization Layer*, welches die Laufzeitumgebung des virtualisierten und entfernten Desktops mit der Maschine eines MobiDesk Kunden interagieren lässt, werden Informationen und durch den

Anwender abgesetzte Befehle verschlüsselt zwischen beiden Parteien übertragen. Im gegensatz zu damals etablierten Technologien wie VMware, welche das komplette Betriebssystem in Form von VMs abstrahieren, agiert MobiDesk auf der feingranulareren Ebene von Sitzungen. Demnach können mehrere Benutzer auf der gleichen Hardwarekomponente eines Dienstes aktiv sein. Die lokal betriebene Client Anwendungen ist für die Isolation der Benutzer untereinander verantwortlich. Die Entkopplung der Sitzungen von deren Laufzeitumgebungen ermöglicht, mit der Migration aktuell verwendeter Benutzerdaten auf eine andere Maschine, Wartungsarbeiten an beliebigen Virtual Desktop Servern. Die in Kapitel 5 vorgestellte Architektur ist prinzipiell um die Orchestrierung von Sitzungen erweiterbar. Eine derartige Umsetzung verlangt einen direkten Kommunikationskanal von spezieller Software in für die Bearbeitung einer Aufgabe bereitgestellten VMs mit dem Brokersystem und ist mit der in Kapitel 5.5 *FeedbackServer* Struktur realisierbar.

Mit *virtual desktop computing* verschiebt sich der Großteil von Berechnungen und Speicherkomponenten von den Geräten eines Endanwenders auf das Netzwerk eines CSP . Die Bereitstellung von Remote Desktops als Cloud Dienst, soll dabei eine Interaktion mit einer VM von jeglichem Endgerät, welches das *Thin Client Protocol* unterstütz, ermöglichen. Abbildung 8 zeigt eine Architektur zur Bereitstellung von

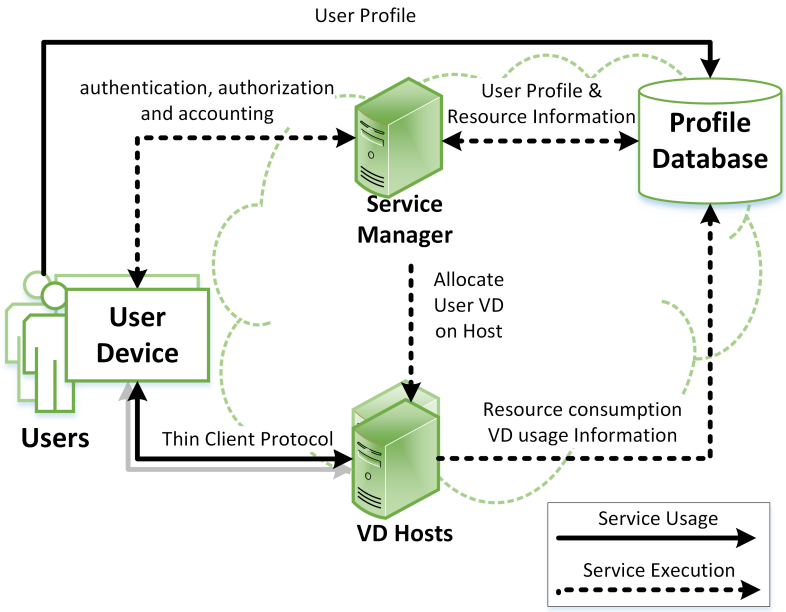

Abbildung 8.: Virtualiserte Desktops in Cloud Computing - Eigene Darstellung in Anlehnung an [DVS+12]

virutalisierten Desktops in einer Cloud Umgebung. Etablierte *Thin Client Protocols* wie Remote Desktop Protocol (RDP), Simple Protocol for Independent Computing Environments (SPICE) oder Virtual Network Computing (VNC) ermöglichen den Benutzern eines virtuellen Desktops auch den Betrieb von legacy Applikationen. Mittels interaktiver Kommunikation zwischen Endgeräten und dem Service Manager, kann

der angefragte Verbindungsaufbau zu der jeweils passenden Maschine bzw. deren *Thin Client Protocol* Server weitergeleitet werden[1]. Für diesen Mechanismus muss der *Service Manager* allen Clients jederzeit ausreichend Informationen über die momentane Auslastung des Dienstes geben. Die Infrastruktur beinhaltet mehrere Virtual Desktop Hosts, auf denen eine Softwaresammlung installiert ist und ein visualisierender Endpunkt zur Übertragung des Bildschirminhalts eines Hosts zum Endgerät des Benutzers betrieben wird. Bei erfolgreicher Authentifizierung eines Users mit dem *Service Manager*, wählt dieser den für die Anfrage passenden Host unter Betrachtung der in der Datenbank abgelegten User Profilen und Informationen über die Hardwareressourcen aus. Der *Service Manager* agiert in dieser Konstellation als Vermittlungsschicht zwischen dem Anwender des Dienstes und den bereitgestellten Ressourcen. Durch Optimierungen der Zustellung des Services wie beispielsweise der automatischen Skalierung bereitgestellter Ressourcen, kann dynamische Last mit Cloud Mechanismen unter dem *pay-per-use* Modell zur Kosteneinsparung eines Dienstes beitragen [DVS+12]. Die Definition von Anwendungsprofilen von auf den virtualisierten Desktops agierenden Diensten kann in einem solchen Szenario zur effizienten Vermittlung von Anfragen an Maschinen mit freien Ressourcen beitragen. In der vorliegenden Arbeit wird auf die Definition eines *User Device* verzichtet, die grundlegenden Prinzipien des *Service Manager* decken sich jedoch teilweise mit denen des erarbeiteten Brokersystems.

Mit Systemen wie der Mandantenfähigen FOSS-Cloud werden virtualisierte Gastsysteme auf einem zentralen Knoten zur Verfügung gestellt [Bra12]. Dieser *Master Node* beinhaltet eine Benutzerschnittstelle für CMS Funktionalitäten. Die auf mehreren *Data Nodes* platzierbaren VM Instanzen sind mit der zentralen Managementkomponente über Switches verbunden. Dem Anwender der FOSS-Cloud wird ein *"Gastfenster"* über die SPICE Technologie von dem virutalisierten Server bereitgestellt. In Templates festgehaltene Eigenschaften über die Hardware einer zu erstellenden VM sind über die Web-basierte Management Konsole definierbar. Dabei können die Anzahl an Prozessoren, RAM, Speicher, UTC, und das Betriebssystem deklariert werden. Vorgefertigte "Golden Images" einer bestimmten Benutzergruppe können mit spezieller Software versehen werden und beispielsweise Bi-direktionales Audio und Videoströme verarbeiten. Ebenfalls können sich Benutzer durch Smartcard Technologien mit dem System bzw. virtualisierten Desktops Authentifizieren. Die USB Weiterleitung von der lokalen Maschine zu einer durch FOSS virtualisierten VM kann durch die SPICE Konsole realisiert werden [fos]. Die in der vorliegenden Arbeit vorgestellte Architektur schreibt keine Einschränkungen bei der Wahl einer CMS ähnlichen Technologie vor, sondern nutzt vielmehr deren öffentliche Schnittstellen zur Umsetzung von Pro-

[1] Im Gegensatz zu *Batch Processing*, in dem die jeweilige Aufgabe bei hoher Auslastung der Infrastruktur für eine spätere Bearbeitung vorgesehen wird, basiert die Aufgabenumsetzung durch den Service Manager auf Echtzeitdaten.

blemstellungen. Eine Hardware-basierte Authentifizierung ist in der Architektur nicht vorgesehen, kann jedoch manuell um diese erweitert werden.

Anwendung von Software in virtualisierten Umgebungen

Die Bedürfnisse eines CSC variieren je nach Problemstellung. Gelten hohe Anforderungen einer bestimmten Software an physische Ressourcen der ausführenden Maschinen, ist deren lokaler Einsatz bei dem Endanwender monetär unattraktiv. Gleiches gilt bei Anwendungen, deren Lizenzkosten für den dauerhaften Eigengebrauch nicht tragbar sind. Für ein an diesen Gesichtspunkt angelehntes Szenario wird auf Abschnitt 6.4.1 verwiesen.

Langmann et. al. diskutierten in [LA12] verschiedene Herangehensweisen an *multiuser remote access* auf 3D Simulationssysteme in virtuellen Laboren. Mit Virtual Private Network (VPN) Zugängen auf die Infrastruktur einer Universität sind Studenten in der Lage deren Lizenzmodell auf bereitgestellten Maschinen zu verwenden. Dabei kamen zur Bildschirmübertragung Technologien wie VMWare, TeamViewer, VirtualBox und Windows Remote Desktop auf den die Software ausführenden Maschinen zum Einsatz. Der betrieb klientenbasierter Anwendungen kann immer nur im Rahmen der Hypervisor-Hardware und programmatischer Performanz agieren, weswegen die maximale Anzahl an Benutzern einer Maschine definiert werden muss. In verschiedenen Versuchen änderte sich je nach aufkommender Rechenlast auch die Antwortzeit der virtualisierten Desktops, worunter die Benutzbarkeit des virtuellen Labors litt. Letztendlich wurden drei physische Maschinen mit vorinstallierten 3D Applikationen und hochwertigen Grafikkarten im Servermodus betrieben. Der entfernte Zugriff auf virtualisierte Desktops wurde den Benutzern dabei über verschiedene Ports der jeweiligen Technologien bereitgestellt. Mit Konzepten der in der vorliegenden Arbeit definierten Architektur ist, wie in Abschnitt 6.4.1 beschrieben, auch ein solches Anwendungsszenario umsetzbar.

Ku et. al. beschrieben in [KCC+10] eine Methode zur Verteilung, Ausführung und dem Management von angepassten Applikation. Der Fokus lag dabei auf Prinzipien der Virtualisierung von Software, welche die Organisation von Anwendungen und Daten in virtuellen Schichten verwaltet. Unter Kooperation der in einem *Management*- sowie *Execution Module*, führt die untergebrachte Virtualisierungslogik entsprechende Operationen auf dem lokalen Dateisystem und einem zentralen *Application Repository* aus. Dabei wird eine Anwendung mit dem *Execution Module* aufbereitet, woraus eine virtualisierte Applikation extrahiert und die Spätere Ausführung im *Repository* abgelegt wird. Durch Repositorien eines *Distribution Server* ist dem Anwender die Auswahl und Ausführung der virtualisierten Applikation von seiner lokalen Maschine aus ermöglicht. Die Diskussion dieser Arbeit dreht sich um unterschiedliche Modu-

le bzw. Pakete, die je nach Szenario oder der zu virtualisierenden Anwendung den Betrieb verschiedener Technologien verlangen. Beispielsweise benötigt eine Banken-Software security & authentication Module. Eine auf die Organisation abgestimmte Web-Entwicklungsumgebung kann hingegen von netzwerkbasierten Database Management System (DBMS) und diversen Debugger Anwendungen abhängig sein. Das komplexe Setup der voneinander isolierten Applikation ist im Prozess der Virtualisierung automatisierbar. Die Rechte, mit denen eine virtualisierte Anwendung ausgeführt wird, können für einen Anwendungsfall auf benötigte Operationen eingeschränkt werden. Treten Fehler in der virutalisierten Software auf, kann diese in das Repository als neue aufgebaute und lauffähige Version eingepflegt werden. Die Prinzipien der Softwarevirtualisierung sind Bestandteil der Vorliegenden Arbeit. Die *Management-Execution* Module sind nur in einem übertragenen Sinn mit dem Brokersystem verwandt. Des Weiteren findet eine potentielle Problemlösung nicht auf Applikations- sondern Plattformebene statt und ist anstatt einer direkten Ausführung mit infrastrukturellen Manipulation einem CMS verbunden. Der in Abschnitt 4.4 gezeigten Technologieen entspringende *Cloud Images* bergen jedoch Parallelen mit den im *Application Repository* virutalisierten Anwendungen.

Die Virtualisierung von Anwendungen ist in vielen Fällen durch *Sandboxing* zu verwirklichen. Die in [Inc13] beschriebenen Vergleiche weisen deutlich auf die Anfälligkeit und die Schwächen bei dem Betrieb von *Sandbox*-basierter Software hin. Grundlegend vermitteln *Sandboxes* ausgeführte Aktionen durch einen nicht vertrauenswürdigen *User-Space* an den Operting System (OS)-Kernel. Dabei besteht die Einschränkung bzw. Weiterleitung von Application Programming Interface (API)-Aufrufen der *Sandbox*-Anwendung aus einem Filter über die entsprechenden Systemoperationen. Findet eine *Malware* eine Schwachstelle innerhalb der *Sanbox*, ist das Host-System anfällig für Kompromittierungen. Nicht vertrauenswürdige Aufgaben werden daher innerhalb einer virtualisierten und von der Produktivumgebung isolierten Hardware ausgeführt. Ausgeführte Systemoperationen eines Gastsystem können unter Verwendung von durch ein CMS betriebenes Echtzeitanalysewerkzeugen auf Böswilligkeit geprüft und blockiert werden. *Arbeiter-VMs* sind mit der Ausführung von Anwendungen (sogenannten *Tasks*) betraut, deren Auswirkungen auf die VM überwacht werden. *Sandbox*-basierten Technologien sind im Gegensatz zu Virtualisierungstechnologien und damit ermöglichten Isolation von *Arbeiter-VMs*, nicht in der Lage, die Verhinderung von Eingriffen in Netzwerk zu garantieren. Darunter zählen Domain Name System (DNS)-Anfragen, Versuche ein direkt adressiertes Datagramm an das Internet zu senden, Prozesse, die durch diesen Task gestartet wurden, Lese- und Schreibaktionen auf *raw devices*, System Calls und registry access. Zusammenfassend sind Malware und vergleichbar destruktive Implementierungen durch eine Auslagerung in eine virtualisierte Umgebung leichter analysierbar. Die weitere Abgrenzung von Produk-

tivsystemen durch den Hypervisor begünstigt das Verhindern einer Ausbreitung und die Echtzeitanalyse infizierter Systeme bei anhaltenden Angriffen. Joe-Uzuegbu et. al. geben in [JUIE15] eine Einführung in Techniken für virtualisierte *Malware Forensic* im Bereich des Social Engineering. Die geführte Diskussion bestärkt die in [Inc13] getroffenen Aussagen im Bezug auf die Stärken der Isolation von virtuellen Ressourcen. Verdächtige Dateien, die beispielsweise als Email Anhang empfangen wurden, sind in einer solchen Umgebung leichter zu identifizieren und analysieren. Das in Kapitel 5 beschriebene System fokussiert sich nicht auf Sicherheitsaspekte und hat keinerlei Bezug zu Sandboxing. Während sich die in [Inc13] Aufgeführten Aspekte auf Windows-basierte VM bezogen, ist die Erstellung einer vorkonfigurierten Forensikinfrastruktur mit den in Abschnitt 4.4 beschriebenen Technologien ein mögliches Anwendungsszenario des Brokersystems.

3.2. Cloud Broker Systeme und Architekturen

Nach [Ent15] ist der Erfolg eines Cloud Service Broker von den in Abbildung 9 dargestellten Faktoren abhängig. Ein strategischer Plan beinhaltet operationale Anforde-

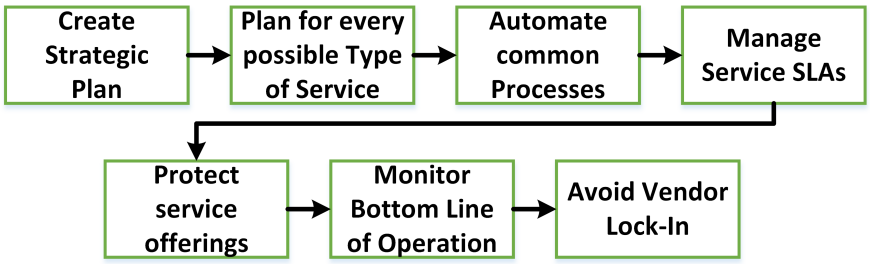

Abbildung 9.: Erfolgsfaktoren von Cloud Service Brokern nach [Ent15]

rungen an die Infrastruktur der Vermittlungsschicht und muss in jedem Fall skalierbar gehalten werden.Die Abbildung von Geschäftsanforderungen auf anzubietende Dienste ist mit einer problemspezifischen Analyse und Abwägung des verwendeten Cloudmodells bzw. CSP verbunden. Das Brokermodell sollte daher alle Möglichkeiten der Erweiterung unterstützen und in der Lage sein IaaS, PaaS und SaaS Mechanismen bereitzustellen. Mit der Automation von infrastrukturellen Aufgaben, individuellen Prozessen sowie der Auslieferung eines Dienstes durch mehrere Arbeitsschritte zusammenfassende Benutzerschnittstellen, verringert sich der Wartungsaufwand eines Cloud Service Brokers. Die Infrastruktur der Vermittlungsschicht ist ausschlaggebend für die Erfüllung von beworbenen SLAs und muss jederzeit für einen Abgleich mit diesen messbar sein. Mit einer zentralen Autorisierungsstelle des Brokers und der Anwendung von rollenbasierten Sicherheitskonzepten ist der Dienst gegen unbefugten Zugriff zu schützen. Des Weiteren sind *accountability* Eigenschaften, *desaster recovery* Mechanismen und die Übereinstimmungen mit angegebenen Standards, einzuhalten und

dem Kunden der Vermittlungsschicht für die Erschaffung einer Vertrauensbasis offen zu legen. Die Überwachung der Betriebskosten eines Brokers ermöglicht ein *pay per use* Kostenmodell des Dienstes. Auf Grundlage dieser Informationen kann eine Einschätzung über die Rentabilität des Dienstes bei Betrieb auf der Infrastruktur eines externen CSP stattfinden. Unter Verwendung von Industriestandards oder open source Technologien, die mehrere Hypervisor und Betriebssysteme unterstützen, kann die Vermittlungsschicht als eine von Cloud Anbietern unabhängige Applikation betrieben werden. Während dieses in sieben Module unterteilte Rahmenwerk allgemeingültige Ratschläge bei der Erstellung eines Brokersystems erteilt, stellt die vorliegende Arbeit ein Werkzeug zur effizienten Umsetzung einer explizit ausformulierten Unternehmung dar und berücksichtigt daher nicht alle aufgeführten Faktoren.

Das in [LCX12] von Liu et. al. vorgestellte *CompatibleOne* Brokersystem beschäftigt sich mit dem Deployment und Management jeglicher Typen von existierenden Cloud Services. Diese können von unterschiedlichen Anbietern stammen, deren Dienste anhand von SLAs mit *CompatibleOne* ausgehandelt wurden. Das Hauptaugenmerk dieses Brokers sind daher *Service Delivery Platform* Funktionalitäten. Durch ressourceneffiziente virtualisierung sind *"Green Cloud"* Infrastrukturen umsetzbar, die auf umweltfreundliche Art und Weise betrieben werden können. Mit dieser Architektur ist die Erstellung, das Deployment, sowie das Management von öffentlichen, privaten und hybriden Cloudplattformen möglich. Das open source Projekt hilft Entwicklern bei der Definition und dem Management der auf die jeweilige Anwendung und den CSP passenden Laufzeitumgebung und trägt zur Vermeidung von *vendor lock-in* Problemen bei. Von *CompatibleOne* wird eine vollwertige Laufzeitumgebung zu Verfügung gestellt, die von CSPs integriert werden kann. Damit wird der vom CSP angebotene Dienst von dem indirekten Zugriff eines CSC auf die *Service Delivery* Schnittstellen des Brokers entkoppelt. *CompatibleOne* baut auf der *ACCORDS* Plattform auf, die auf dem Zusammenspiel von 4 Modulen basiert, die unabhängig voneinander aufgesetzt werden können und damit eine horizontale Skalierbarkeit des Brokers ermöglichen. Anforderungen für Bereitstellung von Ressourcen werden in einem *CompatibleOne Request Description Schema* definiert und in eXtensible Markup Language (XML) Format an das System weitergereicht. Diese grundlegende Definition eines Dienstes hat Konfigurationsdateicharakter und repräsentiert dessen *MANIFEST*. Dieses wird vom System in einen *resource provisioning PLAN* übersetzt und an die eigentliche Broker Technologie weitergereicht. Der Broker übernimmt die Vermittlung von Bereitstellungsoperationen. Die Platzierung einer zu vermittelnden Ressource hängt von registrierten CSPs und deren SLAs mit dem Brokersystem ab. Die Bereitstellung von Ressourcen oder Applikationen geschieht auch unter Verwendung der im MANIFEST deklarierten Anbietern, sowie der Betrachtung derer momentaner Auslastung. Wenn eine Deployment-Anfrage an den Anbieter eines Dienstes in einer Fehlernachricht re-

sultiert, findet das System den nächstbesten CSP mit vergleichbarem Manifest und sendet das Request an diesen. Innerhalb des *MANIFEST*s können auch *Energy Monitoring* Parameter für eine Green IT Verteilung von Ressourcen formuliert sein. Im Gegensatz zur vorliegenden Arbeit bietet dieses System eine Dienstschnittstelle für CSCs und CSPs an. Ebenfalls stützt sich diese nicht auf *ACCORDS*, mit der Anwendung der *Thrift* Technologie ist ein entsprechender Broker dennoch modular verteiltbar und skalierbar.

Die Kontrolle über virtualisierter Anwendungen wurde von Kim et. al. in [KKR+12] durch das Allokationsschema von VMs mit einem experimentellen Cloud Broker zur maximalen Ausnutzung von Ressourcen und der daraus resultierenden Verringerung der Kosten argumentiert. Die Vermittlungsschicht besteht dabei aus zwei logisch getrennten Ebenen. Innerhalb der *Workflow Management* Schicht sind dem Anwender Möglichkeiten zur Komposition von Workflows mehrerer Dienste in Abhängigkeit gewünschter SLAs gegeben. Eine darin enthaltene *Workflow Scheduler* Komponente ist für die Interaktion mit der *Resource Management* Schicht verantwortlich, die neben der Erstellung von VMs auch ein Anfrage- und Ressourcenmonitoring betreibt. Die Allokation von VMs innerhalb eines *Resource Pool* kann auf historischen Daten basieren und beispielsweise die Einhaltung von SLAs, Lastverteilung und weitere Attribute in die Auswahl eines passenden CSP mit einbeziehen. Die in Kapitel 5 gezeigten Strukturen beziehen sich vielmehr auf dynamische Eingangsdaten und der Verarbeitung von Abhängigkeiten in vorkonfigurierten und virtualisierten Plattformen, anstatt der Definition von Workflows auf Ressourcenebene. *Load Balancing* Aspekte werden dabei, sofern nicht ausdrücklich formuliert, auf den jeweiligen CMS übertragen.

Spillner et. al. beschrieben in [SBBS12] einen ökonomisch getriebenen Ansatz zur Steigerung der Granularität von reservierungsbasierten Rechen- und Datenhaltungsvirtualisierungen. Als Motivation wurde die unbenutzte Rechenleistung von bereitgestellten Ressourcen in den resultierenden *Overhead* übersetzt. Dabei wird die inaktive Rechenleistung einer Ressource in Minuten u betrachtet, mit einem stündlichen Kostenanschlag c assoziiert und unter Beachtung des prozentualen Mehraufwand bei der Bereitstellung von VMs mit der Hypervisortechnologie o in einen *Overhead* $U = (\frac{u}{60} * (1 - o) * c)$ übersetzt. Bei $u = 40min$, $o = 5\%$ und $c = 10$€ ergibt sich ein Wiederverkaufspotential der Ressource von $U = 6,33$€. Abbildung 10 zeigt den beispielhaften Aufbau einer Verschachtelten Cloud anhand Basis-VMs. Die in *Level L2* bereitgestellten *sub-VMs* ist Grundlage der Verteilung von ungenutzten Ressourcen einer sich auf *Level L1* befindenden Basis-VM eines Benutzers. Eine Rockkopplung der *L1* Ebene zur Brokerstruktur stellt ungenutzte Ressourcen in einem *Marketplace* bereit. Innerhalb dieses Dashboards sind Regelwerke über die Nutzung bzw. Teilung von VM-Ressourcen durch deren Besitzer konfigurierbar. Für das Anbieten einer Ressource an weitere Benutzer des Brokers, sind mehrere Optionen wie deren

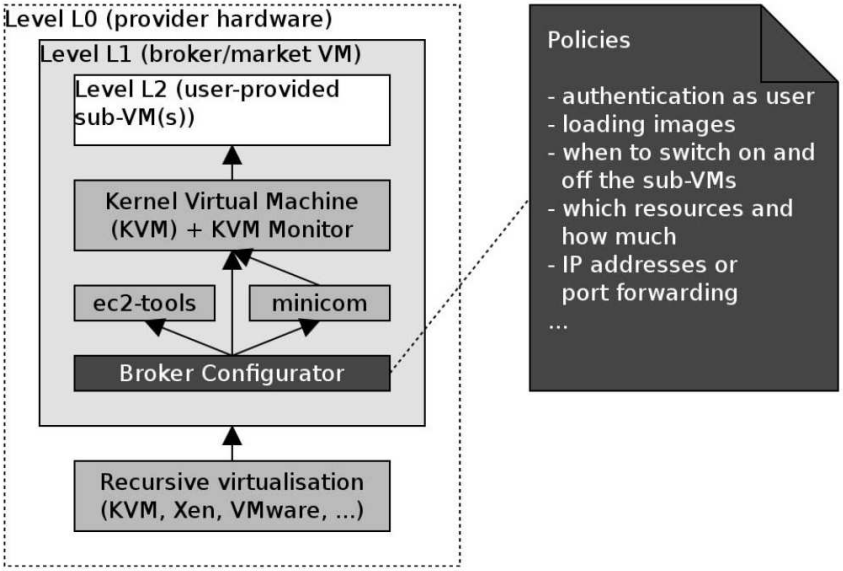

Abbildung 10.: Verschachtelte Cloud mit Basis-VMs [SBBS12]

Wiederverkauf, Bereitstellung innerhalb einer Community und benutzerspezifischen Reservierung, möglich. Der Vorschlag eines Zwischenliegenden Cloud Resource Broker, der zwischen mehreren Anbietern vendor-neutral agiert und als Agent zu Gunsten des Users betrachtet wird, benötigt in jedem Fall administrative Fähigkeiten auf *L1* Ebene aller involvierten VMs. Der Overhead durch die Reservierung von verschachtelten Ressourcen ist bei kleineren Berechnungen unbedeutend, in der Masse kann dadurch aber ein hohes Maß an *overprovisioning* stattfinden, was in einer Kostensenkung auf Benutzerseite resultieren kann. Eine verschachtelte Virtualisierung ist in der vorliegenden Arbeit nicht vorgesehen. Die Anwendung dieses Prinzips kann jedoch, sofern durch das CMS unterstützt, in die Verarbeitungsplattformen mit einbezogen und durch die Brokerstruktur verwaltet werden.

In [AK14b] zeigten Awasthi et. al. einen Brokermechanismus, dessen Hauptaugenmerk auf der Identifikation von *frequent clients* basiert. Dabei wird ein *alocation mechanism* genutzt, der Anwender bevorzugt, die häufig mit Broker interagieren. Weniger aktive Benutzer des Brokerdienstes müssen mit einer höheren Wartezeit rechnen. Mit der Analyse von Benutzeranforderungen an einen Dienst identifiziert der Broker verschiedene, diese Eigenschaften erfüllenden, CSPs. Die Ressourcen der Vertragspartner einer solchen Vermittlungsschicht sind daher in den meisten Fällen unter hoher Last, was deren Erlös steigert. Bei *frequent clients* verringert sich im Normalfall die Wartezeit auf einen Dienstes. Dieser Vorteil resultiert aus der Vertrauensbasis der Benutzer zur Vermittlungsschicht, anstatt der direkten Interaktion mit einem, den gewünschten Dienst anbietenden, CSP. In [AK15] wurde diese Architektur um einen *pricing* Aspekt für die Dienste involvierter Anbieter erweitert. Mit Angabe der *reliability* und *availability* eines Dienstes, sind CSCs zu einer feingranulareren Auswahl von

koordinierten Ressourcen im Stande. Die vorliegende Arbeit deckt keinerlei Modellierungsmöglichkeiten der Anforderungen an Ressourcen durch den Endanwender ab. Derartige Details sind durch den Administrator eines Systems an zentraler Stelle zu konfigurieren und auf den jeweiligen Plattformtyp abzustimmen.

Kim et al. nahmen in [KHK+14] Bezug auf den *Reservation-Based Cloud Service Broker*, der als Zwischenschicht von CSC und VMs verschiedener *Clouds* fungiert. Dabei werden eingehende Anfragen analysiert und in ein Applikationsprofil übersetzt, das ebenfalls von gleichartigen Anfragen genutzt werden kann. Des Weiteren existieren separate VM-Pools für *active* und *idle* VMs, deren Anzahl mit der momentanen Aufgabenlast skaliert. Das System basiert auf Reservierungs-, Monitoring und Allokationsmodulen, die in weitere Komponenten unterteilt sind. Ein bemerkenswerter Teil des Reservierungsmechanismus ist die Vorhersage benötigter Ressourcen, mit der Latenzen von Instanziierungen bis zur endgültigen Benutzung von VMs präventiv beseitigt werden können. Nicht nur die VMs an sich, sondern auch die darin enthaltene Applikationen werden für ein aussagekräftiges Monitoring der *Clouds* analysiert. Neben der Allokation von VMs mit *Jobs*, können die ausführenden VMs nach Bedarf mit zusätzlichen Ressourcen bestückt werden. Alle Interaktionen in diesem System sind durch ein *Securtiy & Authentication Management Module* abgesichert. Der Fokus der vorliegenden Arbeit liegt weder auf der Interaktion unterschiedlicher Clouds, noch auf Prinzipien der Reservierung von Ressourcen. Beide Aspekte sind jedoch bei entsprechender Formulierung von Anwendungslogik in das Brokersystem integrierbar. Die Bereitstellung mehrerer Anwendungen und deren Organisation in Profilen grenzt sich durch die dynamischen Anfragen der Endanwender und der damit resultierenden Erweiterung um neue Anwendungsprofile von der den statischen Metadaten über verschiedene Plattformen ab.

Chatterjee et al. identifizierten in [COA+14] die unabhängige physische Verteilung von Ressourcen in der Cloud Computing Domäne als eines der gravierendsten Konzepte zur Steigerung der Quality of Service (QoS). Die in einer Cloud typischerweise angewandten Hintergrundaktionen wie Ressourcenallokation, Lastverteilung oder *load sharing* Technologien, sind für den Anwender hinter einer abstrahierten Schnittstelle zu verbergen. Durch Anwendung des *Conductance* Algorithmus lässt sich die *makespan* von VMs und Ausführungszeit von *Cloudlets* verbessern. Das in Kapitel 5 beschriebene System gibt die Umsetzung der QoS ohne Angaben über Benutzeranforderungen an das jeweils verwendete CMS weiter.

Amshavalli et. al. stellten in [AK14a] ein auf semantischen Technologien beruhendes Brokersystem vor. Der Semantic Cloud Resource Broker (SCRB) fungierte dabei als Vermittlungsschicht zwischen Eucalyptus- und OpenNebula Private Clouds und ist mit der Steigerung der Verfügbarkeit von Cloud Ressourcen verantwortlich. Ein *ontology-indexing* und darauf angewandte *retrieval* Algorithmen sind das Herzstück

dieser Architektur. Jede *cloud middleware* implementiert in gewissem Maße eigene Mechanismen und Protokolle. Eine Brokerstruktur ist daher mit der Vermeidung von Interoperabilitätsproblemen bei dem Ressourcen-Management heterogener CSPs und deren CMS zu argumentieren. Die Semantik der CMS Systeme wird für das Finden geeignetster Ressourcen verwendet. Bei negativen Suchergebnissen ermöglichten alternative Einstellungen der verwendeten precicion und recall einer Anfrage die Auflistung verwandter Ressourcen. Die Anwendung der in Kapitel 5 beschrieben Architektur verlangt die spezifische Formulierung von Aktionen und deren individuelle Implementierung, weswegen bei der Entwicklung tiefgreifende Kenntnisse über den jeweiligen Hypervisor oder CSP unumgänglich sind.

In [GGB+15] diskutierten Guzek et. al. damalige Praktiken von Cloud Broker Systemen und identifizierten zukünftige Herausforderungen in diesem Bereich. Grundlegend verbindet ein Cloud Service Broker die Wünsche und Bedürfnisse von CSCs mit der tatsächlichen Umsetzung von Diensten auf der Infrastruktur eines CSP. Als *State of the Art* sind die Aggregation von Diensten mehrerer CSPs, sowie deren Integration in einen Eigenen Dienst aufgeführt. Bei der Integration besteht ein gemeinsames System aus öffentlichen und privaten Clouds mehrerer CSPs, die sich gegenseitig nutzen oder auf einander aufgebaut werden können. Der Spezialfall eines Cloud Brokers wird als *customization* bezeichnet. Diese Kombination aus Aggregation und Integration von Cloud Diensten ist meist in der Brokerplattform implementiert. Die Sicherheit eines komplexen Systems ist immer nur so stark, wie die Sicherheit des schwächsten, für den Betrieb des Systems relevanten, Elementes. Bei der Verarbeitung sensibler Daten muss daher die Sicherheit aller abhängigen Dienste gewährleistet sein. Die Umgebungen eines Cloud Service Broker ist je nach Verwendungszweck mit stetigen Veränderungen der Infrastrukturn und Servicekatalogen involvierter CSPs konfrontiert. Ein weiteres Forschungsgebiet ist daher die Organisation multipler Softwareagenten, die sich dynamisch reorganisieren können. Die aufgeführten Herausforderungen bei Verwendung von Cloud Diensten in einem Broker können auch für das in Kapitel 5 beschriebene System nicht ausgeschlossen werden.

Das von Corradi et. al. in [CFP+15] vorgestellte *Cloud4SOA* Brokersystem ist als Zwischenschicht von Anwendungsentwicklern und PaaS Anbietern anzusehen. Da noch keine allgemein akzeptierten Einigungen und Standardisierung im Bezug auf PaaS Ontologieen existieren, nutzt *Cloud4SOA* Semantic Web Techniken zur generische Erstellung dieser formalen Darstellung einer Begrifflichkeit. Aus Entwicklersicht sind von unterschiedlichen PaaS Anbietern bereitgestellte Funktionalitäten und Dienste durch eine harmonisierte API der Vermittlungsschicht abstrahiert und programmatisch abfragbar. Für das *Profiling, Discovery, Deployment Migration* und *Monitoring* benötigte Interaktionen mit einem CSP sind dabei als Service Oriented Architecture (SOA) Dienst bereitgestellt. Dies umfasst neben Werzeugen zur Interaktion mit

der jeweiligen Plattform auch das Monitoring und Management von in der Cloud ausgeführten Applikationen. Die Portabilität von Applikationen ist damit zwischen unterschiedlichen PaaS gewährleistet. Aus Anbietersicht rechnet sich die Registrierung mit *Cloud4SOA* durch den Beitritt zu einem cloudbasierten Ökosystem und der damit verbundenen gesteigerten Anzahl von Anfragen an den Dienst. Die vorliegende Arbeit stellt weder eine harmonisierte API noch SOA Dienste für die Interaktion mit der Vermittlungsschicht bereit. Unter Verwendung der in 4.6 aufgeführten Technologie sind jedoch vergleichbare Schnittstellen für den Anwendungsentwickler realisierbar.

Khanna et. al beschrieben in [KJB15] einen, auf dem Konzept verteilter und föderierter CSPs basierenden, *Broker for Common Usage of Resources*. Grundlegend koordiniert diese Vermittlungsstruktur verschiedene *Task*s auf der Infrastruktur verschiedener CSPs. Die *Coordinator* Komponente besitzt privilegierten Zugriff auf Ebene der involvierten Rechenzentren. Für das *Scheduling* und die Platzierung der *Task* werden Statistiken über die Auslastung der Hardware jedes CSPs gesammelt. Mit einer Kombination von Metriken und der Definition von Schwellwerten, sind auslösende Ereignisse durch einen *human manager* zu überprüfen und die entsprechende infrastrukturelle Empfehlung, wie die Migration einer Ressource zwischen zwei Rechenzentren, umzusetzen. Neben einem SLA Management für die Vereinbarung von QoS oder Lasten involvierter *Datacenter*, ist die Architektur des Weiteren in eine *Resource Placement Engine*, sowie *Administrator and Management Engine* modularisiert. Alle Architekturkomponenten können dezentral platziert werden, sind jedoch als logische und zentral agierende Einheit anzusehen. Dieser Punkt ist Bestandteil der modularen Architektur der vorliegenden Arbeit, wohingegen auf die Migration von Ressourcen und Mechanismen zur Steigerung der QoS verzichtet wurde.

Su et. al. diskutierten in [SXFD16] einen Spieletheoretisches Ressourcenallokationsschema für die MediaCloud[2] im Bezug auf mobile Anwender. Mit dieser open source Plattform soll die Forschung an der Verbreitung und Durchsetzung von Ideen oder *Stories* über verschiedene soziale Medien durch einen Broker erleichtert werden. Ein iterativer Algorithmus repräsentiert Prinzipien des *Stackelberg Gleichgewichts*. Das rückpropagieren von Strategien eines schritte ist in diesem Algorithmus unerlässlich, da angewandte Muster alle anderen Schritte beeinflussen können. Die in diesem System eingesetzten Cloud Broker sind nahe an den sozialen Benutzern, bzw. in deren *Social Community* zu platzieren. Unter Einsatz von *high speed communication links* zur MediaCloud Plattform können in dieser, durch in einer Vermittlungsschicht vom Endanwender initiierten, *media tasks* effizient bearbeitet werden. Ist die Menge der auf eine *Community* abgestimmten Broker kleiner wie deren aktive Benutzeranzahl, ergeben sich ökonomische Vorteile. Die von der MediaCloud abhängigen Brokerinstanzen sind daher in der Lage, einen schnellen Datenaustausch mit der Plattform

[2] `https://mediacloud.org`

zu realisieren, was die Antwortzeit eines Dienstes, und das Zeitfenster zur letztendlichen Verwendung einer erworbenen Ressource, verringert. Das wachsende Volumen von Multimediadaten und Verlangen nach QoE der Anwender rechtfertigt die Verwendung des Brokers für mobile Geräte mit limitierten Ressourcen wie deren Bandbreite oder Speichermöglichkeiten. Die Auslagerung der Verarbeitung von Aufgaben in eine Plattform ist ein Prinzip, das in die vorliegende Arbeit übernommen wurde.

Chamorro et. al. diskutierten in [CLPB16] einen genetischen Algorithmus für die dynamische Vermittlung von Cloud Anwendungen. Das Hauptaugenmerk dieses Systems liegt auf der Broker-orientierten VM-Platzierung in Abhängigkeit von Bedürfnissen eines CSC des Brokers. Die dynamischen Anforderungen von Benutzern der Vermittlungsschicht sollen durch Verwendung gleichartiger Dienste heterogener CSPs in ökonomische Vorteile übersetzt werden. Die in einer dynamischen Cloud Umgebung auftretenden und für die Vermittlungsschicht interessanten Probleme sind jeweils aus Sicht des CSP, sowie des konsumierenden CSC zu betrachten. Die Rahmenbedingungen momentaner Angebote der in die Vermittlungsschicht involvierten CSPs können sich jederzeit ändern. Dabei sind die Kosten von Diensten ähnlich dynamisch, wie der durch verfügbare Ressourcentypen bzw. Produkte oder Alleinstellungsmerkmale definierte Produktkatalog eines CSP. Aus Sicht eines CSC können Änderungen an den momentanen Anforderungen an Ressourcen oder in dessen Budget auftreten. Mit Fokus auf die Maximierung der *Total Infrastructure Capacity* und der Minimierung des *Total Infrastructure Price*, sowie *Migration Overhead Costs*, soll die ideale Laufzeitumgebung der VMs eines CSC bestimmt werden. Der genetische Algorithmus beinhaltet unter anderem auch *Not feasible solution reperation*, in denen durch den Benutzer bestimmte Einschränkungen mit verfügbaren Anbietern abgeglichen werden. Die vorliegende Arbeit geht nicht direkt auf die Vermittlung von homogenen Diensten in heterogenen Szenarien ein, die Registrierung einer eigens erstellten Verarbeitungsplattform bei mehreren CSPs können derartige Mechanismen, unter Anfrage der Monitoringsysteme aller Anbieter, realisiert werden.

Roy et. al. diskutierten in [RDAC16, BCF+12] ein QoS basiertes Brokersystem für virtuelle Ressourcen. Die QoS-bewusste Vermittlungsschicht betrachtet unterschiedliche Aspekte von Diensten, wie deren VM Typ, Pricing oder Möglichkeiten zur Wiederherstellung von Daten. Ebenfalls wird Bezug zu Echtzeit Media Streaming genommen. Cloud Provider registrieren ihren Dienst bei dem Brokersystem unter Angabe der nicht-funktionalen Eigenschaften der angebotenen Ressourcen in prozentualer Form. Auf der Anwenderseite ist ein Benutzer in der Lage Kern-Anforderungen an einen CSP zu spezifizieren und über die benötigte Ebene der QoS eines Dienstes zu entscheiden. Eine durch den Benutzer spezifizierte Anfrage beinhaltet alle QoS-Merkmale, welche prozentual ausformuliert wurden. Die Verfügbarkeit einer VM wird im Kontext verfügbarer Netzwerke bzw. Links definiert. Die Verfügbarkeit ergibt sich

aus $A = 1 - (1 - a)^L$ mit a gleich der garantierten Verfügbarkeit des Providers und L der Anzahl Connection links. Die Anzahl Connection links besteht daher aus $L = \frac{ln(1-A)}{ln(1-a)}$ mit $0 \leq A \leq 1$. Die Reliability wird auf Basis der in dem Hypervisor für Virtualisierung verwendeten Hardware gemessen. Die *Scalability* wird anhand eines durch den user gesetzten Workload-basierten Threshold-Wert und der geforderten Elastizität definiert. Zur Laufzeit einer VM wird daher die folgende Annahme getroffen: $\sum_{i=1}^{n} \frac{r_{i_{consumed}}}{r_{i_{allocated}}} > r_{i_{threshold}}$, wobei sich der definierte QoS auf VM-Resource r auswirkt und r_i die i^{te} VM Ressource darstellt. Ist der Wahrheitswert in einem Punkt dieser Metrik negativ, wird eine neue VM mit gewünschter Elastizität gestartet und dem entsprechenden User zugeordnet[3]. Bei einer anschließenden Lastverringerung wird diese VM in den Ressourcenpool zurückgestellt und ein *fallback* zur Ursprungsmaschine durchgeführt. Das System beinhaltet Request- und Responsehandler, die als Schnittstelle zu den Endanwendern der Vermittlungsschicht agieren. Der Request Analyzer betrachtet die Quantität verschiedener Cloudresourcen, um ein Request zu erfüllen. Dieses wird in funktionale und nicht-funktionale Teile übersetzt, die für den Betrieb aller weiteren Module verwendet werden. Das *Cost analyzer and Billing System* berechnet die Basiskosten eines Dienstes, sowie zusätzlich verursachte Kosten des QoS Grades. Ein zentrales Repository beinhaltet verschiedene Anfragen von Benutzern. Alle persistierten Anfragen sind mit einer VM in Resourcen Pool assoziiert. Bei einer neuen, bisher nicht so vorgekommenen Anfrage, wird die entsprechende VM *on-the-fly* erstellt und dem Repository hinzugefügt. Ein *Request prediction System* stellt Vermutungen über zukünftige Anfragen mittels neuralen Netzwerken an, um eine entsprechende Anzahl VMs in Ressourcenpool vorzuhalten. Der *Resource Allocation Manager* ist für die direkte Interaktion mit multiplen CSP bzw. CMS und der Allokation der zutreffendsten VM im Hinblick auf QoS-Anforderungen des User Requests verantwortlich. Die vorliegende Arbeit fokussiert sich auf funktionale Eigenschaften im Bezug auf die Verarbeitung dynamischer Datenquellen und grenzt sich daher stark von den ökologisch getriebenen Mechanismen aus [RDAC16, BCF+12] ab.

Pacini et. al. stellten in [PMG16] einen auf Ant Colony Optimization (ACO) bezogenen Broker Scheduler vor. Parameter Sweep Experiments (PSE) sind Simulationen mit wiederholt wechselnden Eingabeparametern, wobei invalide Parameter das *problem of interest* repräsentieren. PSE ist ein wichtiges Instrument für die Analyse und Entwicklung effizienter *Scheduling* Strategien im Bezug auf die Allokation von *Jobs* in *federate cloud* Umgebungen. Die maximal benötigte Zeit für das Ausführen einer Menge an *Jobs* wird als *makespan* bezeichnet und sollte möglichst gering ausfallen. Das *Scheduling* des Brokersystems wird in drei Schichten umgesetzt. Diese bestehen

[3] Bsp: Allokation einer VM mit 50GB Storage, Threshold von 90% und elasticity von 100%. Beim Erreichen von 45GB wird der VM ein weiterer Storage-Block von 50GB angefügt.

aus einem grundlegenden Broker Level, welches das geeignetste Datacenter für einen *Job* auswählt. Auf Infrastrukturebene geschieht die Allokation von VMs auf der Hardware eines CSP. Innerhalb einer VM wird ein First-In-First-Out (FIFO) basiertes Scheduling der jeweiligen *Jobs* durchgeführt. Inter-datacenter Cloud *Scheduling* bezieht sich auch auf Kombinatorische Optimierungsprobleme, die durch Schwarmintelligenz-Metaheuristiken gelöst werden können und auf der Infrastruktur eines exklusiven CSP ausgeführt werden. State of the Art Strategie ist ACO mit Hilfe derer das Verhalten eines Kollektivs Simuliert wird. Sobald ein Broker Anfragen eines Users für die Erstellung einer Menge an VMs erhält, die das PSE Experiment lösen sollen, wird eine *Ant* für das Finden des geeignetsten IaaS Anbieters instanziiert. Bei einem intra-datacenter scheduling Vorgehen wird für jedes VM-deployment request eines users eine exklusive Ant allokiert. Die Ant-interne *load Information* beinhaltet Informationen über die Central Processing Unit (CPU) Auslastung eines Hypervisors bzw. CSPs, die am Ende zentral abgelegt werden und als Pheromon für andere Ants dienen. Wenn eine Ant von Host zu Host migriert, geschieht dies entweder per Zufallsgenerator oder in Abhängigkeit von der *load Information table* des Hosts. Die Auslastung eines CSP oder Hypervisor spielt in der vorliegenden Arbeit keine übergeordnete Rolle. Die mit ACO entwickelten *Scheduling* Strategien sind auf Grund der Lebensspanne einer Plattform und gegebenenfalls deren Wiederverwendung ebenfalls nicht von Relevanz.

Yang et. al. diskutierten in [YL16] eine Brokerstruktur zur Überwachung der Erdoberfläche anhand der Kombination mehrerer Datenquellen in einem öffentlich zugänglichen System. Das Vereinen multipler Datenquellen verschiedener Agenturen, welche aus unterschiedlichen Attributen und Strukturen bestehen, jedoch in derselben Domäne Aussagekraft besitzen, ist die Vermittlungsaufgabe dieses Brokers. Da es sich bei den zu verarbeitenden Informationen um gigantische Datenmengen handeln kann[4], werden diese in einem gemeinsamen Rechenzentrum verarbeitet. Die durch Satelliten gesammelten Informationen stammen dabei aus unterschiedlich hohen Entfernung und repräsentieren einen bestimmten Orbit, weswegen eine semantische Komponenten für die Übersetzung von Earth Observation (EO) Daten in ein Resource Description Framework (RDF) und der Verkopplung von Ergebnissen mit anderen frei zugänglichen Datenquellen verantwortlich ist. Das implizite *knowledge* wird aus den aufkommenden Daten bestimmt und dem Anwender in Form eines Web Interface visualisiert zur Verfügung gestellt. Die automatisierte Kombination von Datenquellen ist das Hauptaugenmerk des EO Systems und grenzt sich von der Vorliegenden Arbeit durch die fehlende Verarbeitung von Daten auf Benutzerseite ab.

Die Erstellung einer Cloud Föderation wurde von Gupta et. al. [GA16] mit einem vertrauenswürdigem und 'intelligentem' Broker umgesetzt. Dabei lag das Hauptau-

[4] Beispielsweise repräsentiert Chinas *ZY-3* Satellit gesammelte Daten von 300 Tagen in 17 Terrabyte.

genmerk auf der Berechnung der *Reliability* eines CSP anhand historischer Kommunikationsmuster, wie auch Empfehlungen und Erfahrungen von anderen Klienten über diesen. Die Auswahl und Nutzung von fremden Ressourcen basiert auf der berechneten indirekten Vertrauensbasis von CSC und CSP. Möchte ein CSP Teil der Föderation werden, kann er sich bei dem Broker registrieren und bekommt die initiale Rangfolge seiner benachbarten Verbindungen. Die vorliegende Arbeit bietet keinerlei automatisierten Mechanismen zur Aufnahme neuer CSPs in eine Föderation an.

Der von Abderrahim et. al. in [AC16] vorgestellte Broker fokussiert sich überwiegend auf das *Fault Management* im Bezug auf voneinander abhängige Cloud Implementationen. Diese Dienste können auf IaaS, PaaS oder SaaS Ebenen platziert sein und werden mit der darin vorgeschlagenen Architektur integrierbar gemacht

Der von Alsina et. al. in [AIN+16] beschriebene Broker basiert auf Reservierungsmechanismen von in unterschiedlichen Rechenzentren platzierte VMs. Das Vorhalten von virtualisierten Ressourcen an mehreren, geografisch voneinander getrennten Standorten ermöglicht bei entsprechender Verteilung von Aufgaben eine Senkung der benötigten Netzwerklast. Unter Anwendung verschiedener *Scheduler* für die Planung der Platzierung von VMs wurde das *Shortest Request to Cheapest Instance* Prinzip als der für *heavily-loaded low-transfer* Szenarien performanteste identifiziert. Nicht nur die Größe der durch einen Anwender formulierten Anfrage, sondern auch dessen geographischer Standort, wird in die Auswahl des günstigsten Rechenzentrum bzw. CSP mit aufgenommen. In *lightly-loaded low-transfer* Szenarien ist der Einsatz der *Cheapest Instance* Heuristik anzuwenden, bei der die durchschnittliche Antwortzeit, sowie die Kosten für den Datenaustausch und den eigentlichen Dienst, gering gehalten werden sollen. Ein Mechanismus zur geografischen Platzierung von Plattformen grenzt sich, ebenso wie die Anwendung verschiedener Heuristiken, von der vorliegenden Arbeit ab.

3.3. Metamodelle und Domänenspezifische Sprachen

Die von Bauer et. al. in [BMR07] beschriebene dezentralisierte Brokerarchitektur bezieht sich auf die Modellierung und Anreicherung kollaborativer Geschäftsprozesse. Beweggrund der Arbeit waren sich ständig verändernde Geschäftsbeziehungen und sich über mehrere Vertragspartner erstreckende Prozessschritte. Im Kontext des MDSD kann eine derartige Beziehung mit der Deklaration eines Anwendungsmodells abgebildet werden. Unter Verwendung des plattformunabhängigen *PIM4SOA* Metamodells wurde die zentralisierte und verteilter Arbeitsweise des Brokersystems implementiert. Beide Ansätze sind aus einer verarbeitungsunabhängigen Modellbeschreibung unter Definition der *Cross-Enterprise Business Processes* ableitbar. Die zentrale Variante der Vermittlungsschicht kann direkt aus dem Geschäftsprozess abgeleitet werden.

Findet das Management der Geschäftsprozesse nicht auf einer zentralen Maschine, sondern in einem verteilten System statt, muss der Broker Rechte über eine explizite Sicht auf die Gruppierung von Prozessen aller involvierten Maschinen bzw. Systemteile besitzen. Die Bereitstellung der von auf die Domäne der Geschäftsprozesse angepassten Vermittlungsschicht fordert Informationen über Schnittstellen der internen Verarbeitungsmuster aller beteiligten Organisationen. Im Gegenzug verbirgt der Broker die tatsächliche Arbeitsweise der an einem Geschäftsprozess teilnehmenden Partner. Auf Grund der Modularität und Flexibilität der Architektur sind Änderungen an der privaten Implementation eines Prozesse nicht zwangsläufig mit Auswirkungen auf beteiligte Organisationen oder die Vermittlungsschicht verbunden. Die vorliegende Arbeit bezieht sich auf die Anwendung von Prozessen in darauf zugeschnitten Plattformen. Bei Vergabe von isolierten Plattformen an jeden Geschäftspartner ist ein derartiges, über Schnittstellen kommunizierendes System über Rückkopplungen zur Vermittlungsschicht jedoch denkbar.

In [LA14] stellten Lachgar et. al. eine MDSD-basierte Methodik zur Erstellung von Benutzeroberflächen auf mobilen Systemen vor. Unter Einsatz der *XText* und *XTend* Technologien wurde eine domänenspezifische Sprache, sowie diverse Code Generatoren und auf die jeweilige Technologie bzw. Betriebssystem angepasste Transformationen implementiert. Mit der Instanziierung einer allgemeingültigen GUI-DSL, sind unter Anwendung von M2M Transformationen, unterschiedliche Abbildungen der Benutzeroberfläche auf Modelle spezifischer Technologien, wie Android, iOS oder JSF, möglich. Die aus einem Anwendungsmodell generierten Strukturen beinhalten dank der übersichtlichen Grammatik der DSL keinerlei tiefe Verschachtelungen. Die in Abschnitt 5.5 beschriebene DSL beruht auf den in [LA14] aufgezeigten Prinzipien und erweitert die Domäne der Darstellung von Informationen in einer Technologie mit grundlegenden Brokereigenschaften. Die Diskussion einer M2M Transformation ist in dieser Arbeit nicht vorgesehen.

Der von Lethrech et. al in [LEN+14] diskutierte Ansatz der domänenspezifischen Modellierung bezieht sich auf die Kontextsensitivität in dienstorientierten Systemen. Die Aufspaltung von Domänenspezifischen Diensten, dessen Kontext und Geschäftsregeln bietet ein hohes Maß der *Separation of Concerns*. Ein kontextsensitiver Dienst ist von der sorgsamen Analyse und Formulierung aller die Domäne betreffenden Variationspunkte abhängig. Die vorgeschlagene erweiterbare DSL besteht aus fünf, in einem Code Generator vereinten Modellen, die für eine Transformation von Anwendungsmodellen in den Quellcode zu implementieren sind. Dabei ist das bereitgestellte generische Dienste-Metamodell mit dem jeweiligen Metamodell einer Domäne zu instanziieren. Das *domain specific services model* repräsentiert die mit der Domäne assoziierten Dienste, wohingegen mit dem *service variability model* darin auftretende Variationspunkte formuliert werden. Der in einem *domain specific context model* ver-

fasste Domänenkontext, in dem Anwendungsmodelle betrachtet werden, wird durch das *adaptation rules model* mit der Variabilitätskomponente des Systems verbunden. Das *domain specific business rules model* repräsentiert alle durch die DSL ausdrückbaren Geschäftsregeln. Die Analyse des Kontextes eines Systems wird in Kapitel 5 nicht diskutiert, kann aber durch die ausformulierten, automatisch ausgeführten Aktionen der DSL in Abhängigkeit von Eingangsdaten in der Brokerstruktur umgesetzt werden.

In [LBR16] beschrieben Letrache et. al. einen Ansatz zur Erstellung von Key Performance Indicator (KPI)s unter Anwendung des MDA in Online Analytical Processing (OLAP) Projekten. Bestehende Rahmenwerke zur Modellierung von KPIs hatten keinerlei Bezug zu OLAP und wurden zum Anreichen von *Business Strategic Models* oder der separaten Erstellung eines KPI Modells verwendet. Das vorgeschlagene PSM repräsentiert daher die KPIs als Erweiterung des dem *Business Intelligence modeling standard* entsprechenden *Open Information Model* Metamodells. Ein KPI kann grundlegend als multidimensionaler Ausdruck formuliert werden und ist typischerweise auch von *Business Inteligence* Systemen in dieser Form interpretierbar. Aus dem Modellierungsansatz, der mit OLAP Attributen in Beziehung zu setzenden KPIs, sowie deren nachfolgenden Transformation, ergeben sich bequeme Entwicklungsumstände. Die vorliegende Arbeit geht nicht auf die Umsetzung von OLAP Systemen ein.

4. Überblick relevanter Technologien

Der Einsatz unterschiedlicher Technologien ist ein gängiger Bestandteil bei der Implementierung und dem Betrieb komplexer Projekte. Die Kombination verschiedener Technologien, die in einer der Unternehmung entspringenden Anwendung genutzt werden, ermöglicht den Entwicklern die Erstellung mächtiger Architekturen. Durch die Einbettung von API Aufrufen oder der Anwendung von vorgegebenen Mustern von Bibliotheken im Quellcode, sind diese Fremdsysteme bequem in den Ablauf des Systems integrierbar.

Mit den in diesem Kapitel beschriebenen Technologien soll ein grundlegendes Verständnis für die in Kapitel 5 beschriebene generische Architektur, sowie deren in Kapitel 6 geschilderten Anwendung in Verschiedenen Szenarien, schaffen.

4.1. Domänenspezifische Modellierungssprachen

Die modellgetriebene Herangehensweise an die Erstellung einer Software kann textuell oder grafisch durchgeführt werden. Für beide Arten existieren mehrere Lösungen, die im Folgenden kurz vorgestellt werden. Die darauffolgende Diskussion bezieht sich auf die textuellen Grammatiken und die Formalisierung von Sprachen.

Grundlegend bestehen alle Ansätze der domänenspezifischen Modellierung aus einer vierschichtigen Hierarchie. Prinzipiell spezifiziert die jeweilige Schicht dabei immer, wie darauffolgende Instanziierungen auszusehen haben. Wie in Abbildung 11 dargestellt, beinhaltet die *M3* Schicht alle für die Erstellung eines *M2* Modells benötigten Strukturen. Diese Modellierungssprache für Metamodelle wird auch als *Metalanguage* oder *Root Model* bezeichnet. So basieren textuelle Metamodelle auf der Extended Backus–Naur form (EBNF), die als Metasyntax für die Erstellung eigener Gramattiken in *M2* verwendet werden. In der grafischen Variante werden grundlegende Strukturen, Profile und Einschränkungssprachen für die Zusammensetzung in Modellbausteine des Metamodells durch die Meta Object Facility (MOF) bereitgestellt. Die aus einer textuellen Grammatik entstehende Syntax und Semantik des Metamodells folgt hierbei einem Abstract Syntax Tree (AST), der für das Parsen von spezifischen Programmen der *M1* Ebene, deren semantischen Überprüfung und der weiteren Code Generierung verwendet wird. Die in einem bildlichen Diagramm der *M1* Schicht kompositionierten Bausteine stellen ebenfalls ein konkretes Anwendungsmodell dar, dem mit der Object Constraint Language (OCL) Definition des zugehörigen Metamodells eine semanti-

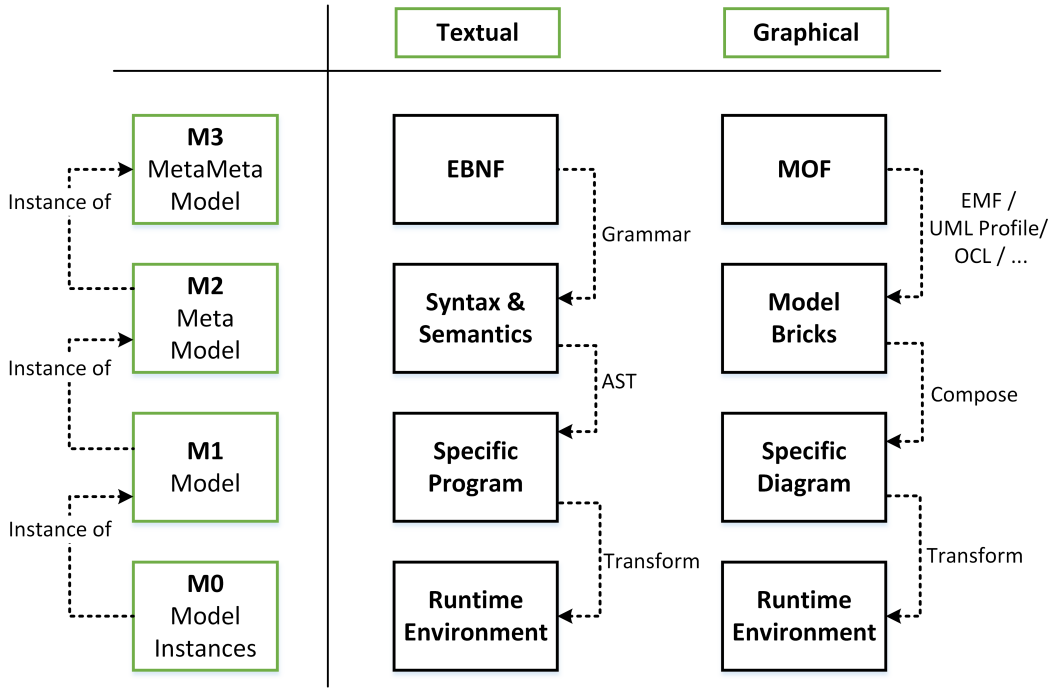

Abbildung 11.: Ansätzte und Hierarchie domänenspezifischer Modellierung - Eigene Abbildung in Anlehnung an [Jö13, SV06, LA14]

sche Bedeutung und Validierung zuzuordnen ist. Die *M1* Ebene bezieht sich dabei immer auf ein konkretes Programm, welches durch Transformationen in unterschiedliche Laufzeitsysteme übersetzbar ist. Die eigens definierte textuelle oder bildliche Grammatik repräsentiert meist Teilbereiche einer spezifischen Domäne und kann innerhalb der *M1* Schicht in einer konkreten Datenstruktur instanziiert werden. Daraus entstehende Programmteile werden unter Verwendung von Codegeneratoren in Textform übersetzt und sind in der Mehrheit der Fälle mit individueller Programmlogik der Problemstellung in den erstellten Komponenten anzureichern. Die unterste Schicht der *M0* Modellinstanzen repräsentiert in jedem Fall ein laufendes System mit existierenden Datenstrukturen, wie beispielsweise Datensätze eines Relational Database Management System (RDBMS) oder aktiven Java Objekten.

Die Definition einer Grammatik kann laut dem Linguist *Noam Chomsky* in dem Vierertupel $G = (V_N, V_T, P, S)$ dargestellt werden. Dabei wird V_N als abgeschlossene Menge nicht-terminierender Symbole betrachtet und steht V_T als abgeschlossene Menge terminierender Symbole gegenüber. Die abgeschlossene Menge der Produktionsregeln P definiert dabei anwendbare Kombinationen von V_N und V_T. Wenn beispielsweise $\alpha \rightarrow \beta \in P$ und $u\alpha v \in (V_N \cup V_T)*$ gilt, dann ist $u\alpha v \rightarrow u\beta v$ eine mögliche Kombination[1]. Das Startsymbol S ist der Einstiegspunkt für das Auflösen eines Ausdrucks und muss in V_N enthalten sein. Weiterhin gilt $V_N \cap V_T = 0$, da eine Regelanwendung von P damit ausgeschlossen wäre. *Chomsky* teilte die Erstellung einer

[1] mit dem $*$ Symbol wird eine 0-n Assoziation ausgedrückt, wohingegen das $+$ eine 1-n Beziehung darstellt.

Grammatik in vier Typen ein. In *Type 0* Grammatiken existieren keinerlei Einschränkungen der Produktionsregeln, was sie zu rekursiv aufzählbaren Sprachen macht. Die kontextsensitiven *Type 1* Grammatiken folgen dem Prinzip von $\alpha A\beta \rightarrow \alpha\gamma\beta$ mit $A \in V_N$ und $\alpha, \beta \in (V_N \cup V_T)*$, wobei A im Kontext von α und β durch γ ersetzt werden kann. Eine kontextfreie *Type 2* Grammatik folgt $A \rightarrow \beta$ mit $A \in V_N$ und $\beta \in (V_N \cup V_T)*$, wobei die linke Seite einer Regel aus nicht-terminierenden Symbolen besteht. Die *Type 3* Grammatiken beziehen sich auf reguläre Sprachen, in denen alle Produktionsregeln links- oder rechtsregulären Strukturen folgen [PYS11].

Die Architektur von Compilern kann unterschiedlich aufgebaut sein, Abbildung 12 zeigt dessen typischerweise angewandten Komponenten. Grundsätzlich soll dabei eine Datenstruktur in eine andere Form abgebildet werden. Die syntaktische Analyse

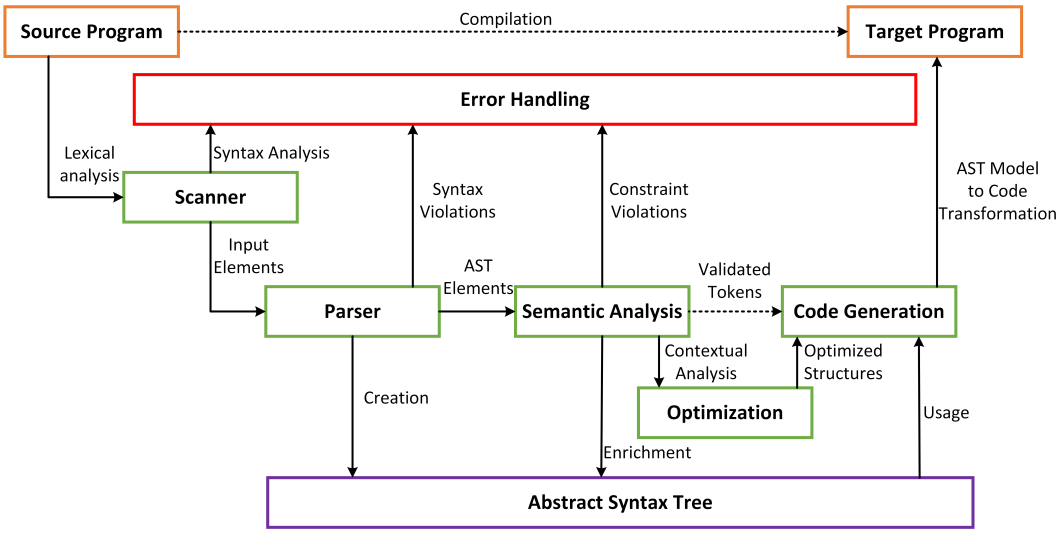

Abbildung 12.: Typische Komponenten eines Compilers - Eigene Abbildung in Anlehnung an [Jö13, SV06, Bet13]

in einem Compiler besteht aus zwei Phasen. Eine *Scanner* Komponente nimmt sich der lexikalischen Analyse an, in der das Quellprogramm in *Identifier, Tokens*, Operatoren, Schlüsselwörter und Zeichensetzung zerlegt wird. Die Grundlagen für dieses Modul sind *Type 3* Grammatiken, wohingegen der *Parser* die syntaktische Struktur mit kontextfreier Grammatik validiert und in einen AST übersetzt. In Abhängigkeit des verwendeten Algorithmus, wie *Top-Down, Buttom-Up* oder *Recursive Decent Parsing*, entstehen unterschiedliche Syntaxbäume. Diese hierarchische Abbildung von Symbolen des Quellprogramms ist in einem anschließenden Schritt mittels einer Grammatiktransformation in eine einheitliche Form zu bringen. Werden bei der syntaktischen Analyse Verletzungen von ausformulierter Symbolik des Quellprogramms gefunden, ist eine entsprechende Komponente des *Error Handling* zu benachrichtigen. Die semantische Analyse der Elemente eines AST bezieht sich auf die Übereinstimmung der Quellstrukturen mit den kontextuellen *Constraints* der Quellsprache. Die Validierung von verschachtelten Konstrukten, die Sichtbarkeit von Variablen und verwendeten Da-

tentypen in den formulierten Programmstrukturen können je nach Kombination eine bestimmte Bedeutung repräsentieren. Die Anreicherung des AST mit der Semantik im Bezug auf einen bestimmten Kontext und aus der Optimierung für die Zielplattform entspringende Strukturen sind Eingangswerte des *Code Generators*. Die Übersetzung des AST in eine das Quellprogramm repräsentierende Anwendung ist oft auf die Definition und Anwendung von *Templates* einer bestimmten Domäne angewiesen.

XText ist ein Eclipse Rahmenwerk zur Erstellung von textuellen kontextfreien Programmiersprachen und DSLs, welches für die in Abschnitt 5.5 aufgeführte Grammatik angewandt wurde. In diesem Framework werden Aspekte einer kompletten Spracheninfrastruktur wie Parser, Code Generator, Interpreter und der Integration in eine Integrated Development Environment (IDE) abgedeckt [Bet13]. Die in der EBNF verfasste Grammatik ist dabei der Grundstein für die infrastrukturellen Komponenten und repräsentiert den AST. Dabei gilt die Zusammenfassung von Produktionsregeln in der Form von $A \rightarrow \alpha_1|..|\alpha_n$ und der zusätzlichen Fragezeichenschreibweise, die eine binäre Entscheidung über ein nicht terminierendes Symbol repräsentiert [PYS11]. *XText* basiert selbst auf dem *Eclipse Moddeling Framework*, um den AST eines Ausdrucks zu erstellen. Für jede in der Grammatik bestehende Regel wird ein EMF-Interface und die zugehörige Klasse generiert, welche alle Features der Regel beinhaltet [Bet13]. Die M2C Transformation folgt dabei jedoch dem in Abbildung 12 dargestellten Informationsfluss. Dabei ist das Quellprogramm als plattformunabhängiges Modell einer in *XText* definierten Grammatik zu betrachten, welches in ein plattformspezifisches Zielprogramm kompiliert wird. Die auf eine Grammatik bezogenen Parser und Code Generatoren sind in diesem Framework ohne weiteres Zutun automatisiert erstellbar. Die syntaktische Analyse eines Anwendungsmodells bezieht sich daher auf die in der Grammatik definierte Symbolik. Anhand des ebenfalls automatisch erstellten AST, kann eine DSL-spezifische Validierung formuliert und bereits während der Implementierung des Anwendungsmodells in der IDE als visuelle Unterstützung integriert werden und zu dessen Optimierung bei der Kompilierung beitragen. Die in der *XTend Template Engine* formulierten Programmschablonen greifen zur Darstellung der instanziierten Grammatik auf die generierten EMF Klassen zurück. Das der Kompilierung entspringende Zielprogramm wird in diesen statischen und typisierten *Templates* auf mögliche Textblöcke heruntergebrochen und ist durch die mögliche Anwendung von Java-ähnlicher Syntax auch mit zusätzlicher Logik anzureichern. Beispielsweise kann der Dateiname einer zu generierenden Klasse von der Verschachtelung in einem Grammatikkonstrukt und Verknüpfungen mit weiteren Regeln abhängig sein und muss daher auf *Template*ebene ausgewertet werden. Die in diesem Rahmenwerk bereitgestellte Entwicklungsumgebung ist zusammenfassend ein mächtiges Werkzeug zur effizienten Erstellung einer domänenspezifischen Sprache. Die Trennung von Domänenwissen und den in *Templates* deklarierten Plattformen, ermöglicht die Aufteilung

des Entwicklungsprozess einer Menge an gleichartigen Anwendungen unter den in Abschnitt 2.3 aufgezeigten Aktoren.

4.2. Cloud Infrastruktur Management mit OpenStack

Die Verwendung von Cloud Technologien ist grundlegend interessant bei Unternehmungen, deren infrastrukturelle Situation von der Last eines angebotenen Dienstes abhängt. Je nach Problemstellung und monetären Begebenheiten ist zwischen Cloudlösungen und der Investition in eigene Hardware abzuwägen. Mit den Anforderungen genügenden Hardware Ressourcen muss für den Betrieb der eigenen Cloud ein softwarebasiertes CMS ausgewählt und konfiguriert werden. Bei Unternehmungen, deren ökonomische Vorteile auf einem bestimmten geografischen Standort basieren, bieten Anbieter wie Amazon mit Diensten, wie der Elastic Compute Cloud (EC2), global verteilte Infrastrukturen zur Auswahl [BKNT11]. Auf Grund des breiten Spektrums der Anwendung von Open-Source CMS Technologien in der Forschung [KL16, HNK+16, BKNT11], wird im Folgenden die OpenStack Technologie vorgestellt. Mit dieser Open-Source Plattform lassen sich skalierbar verteilte Rechenumgebungen erstellen, die dem Anwender die Erstellung eines Systems in öffentlichen, privaten und hybriden Cloudumgebungen ermöglicht [HNK+16].

Logische Architektur von OpenStack

Das OpenStack System besteht aus mehreren sich ergänzenden Teilprojekten, die je nach Anforderungen an die zu erstellende Cloud separat platziert werden und mit einander kommunizieren [HNK+16]. Unter Verwendung von *Publish / Subscribe* Mustern von *Message Queue* Kommunikationsmechanismen, ist die lose Kopplung aller OpenStack Dienste gegeben. In Abbildung 13 sind alle in OpenStack definierten Module dargestellt. Die gestrichelten Rechtecke geben den Geltungsbereich der jeweiligen Komponente an.

Die zentrale Komponente einer OpenStack Architektur ist das *Keystone* Modul, welches unter anderem den *identity service* zur Authentifizierung mit allen, in OpenStack bereitgestellten, Funktionalitäten und der Kommunikation von Modulen untereinander verwendet wird. Eine erfolgreiche Authentifizierung resultiert in einem zeitlich begrenzten Zugang, mit dem der autorisierte Benutzer Funktionen des **CSM!** (**CSM!**) benutzen darf. OpenStack bietet ebenfalls die Möglichkeit der Integration von Lightweight Directory Access Protocol (LDAP) oder Pluggable Authentication Modules (PAM) Systemen.

Um eine horizontal skallierbare und performante *Object Storage* Lösung von CMS-relevanten Datensätzen zu realisieren, wird das *Swift* Modul eingesetzt. Neben Objek-

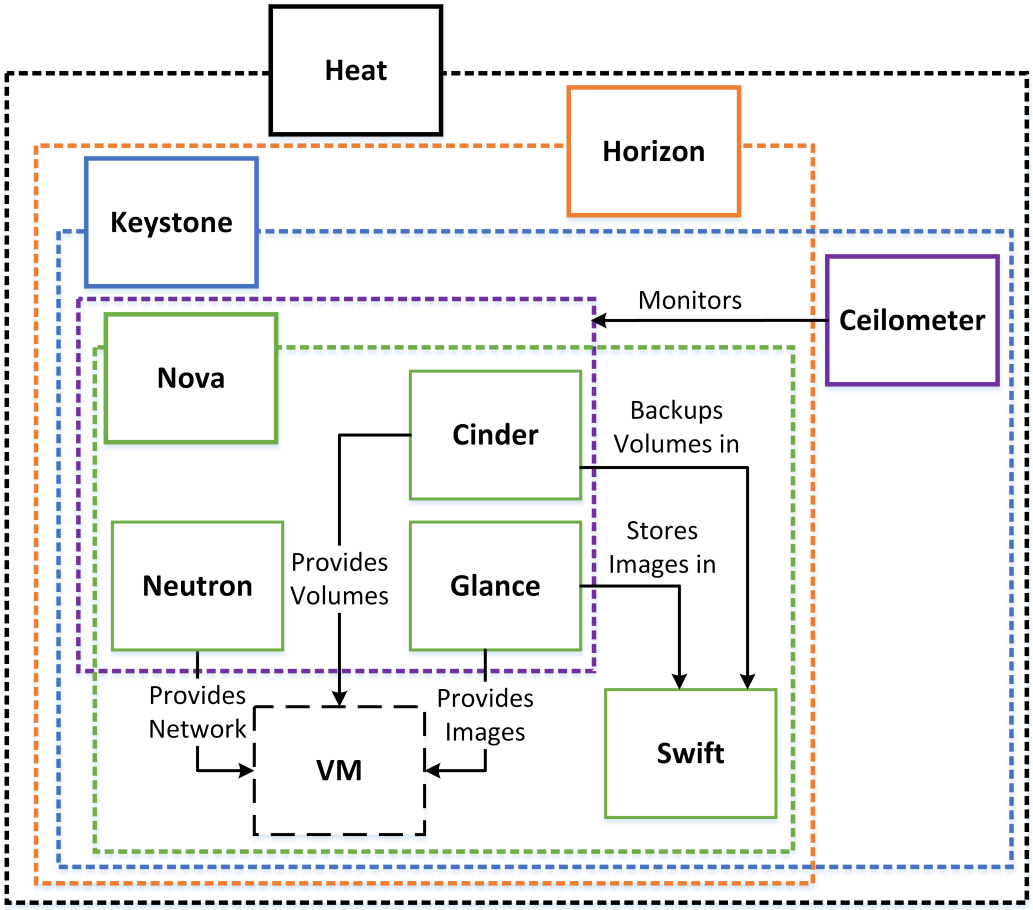

Abbildung 13.: Logische Architektur einer OpenStack Cloud, Angelehnt an [Khe15, FFG+14]

ten wie instanziierbaren *Cloud Images*, befinden sich hier auch Snapshots und Archive von beispielsweise der gesamten Infrastruktur einer Unternehmung zum Zeitpunkt des Erreichens eines Meilensteins. Diese auch auf günstiger Hardware ausführbare Komponente beseitigt Unannehmlichkeiten wie Single Point of Failure (SPOF) Szenarien und kann durch Hochverfügbarkeit von Objekten in automatisierter Weise zur Wiederherstellung fehlerhafter Informationen beitragen.

Alle in *Swift* existierenden Objekte sind innerhalb des *Image Service* der *Glance* Komponente registriert. Durch die lose Kopplung der Metadaten eines bestimmten Objekts von dessen Rohform ergeben sich gesteigerte Möglichkeiten der Orchestrierung von Kernkomponenten. Mit der separaten Datenhaltung werden zudem performantere Abfragen an die jeweilige Technologie ermöglicht.

Die Bereitstellung von persistentem *Block Storage* wird innerhalb OpenStack mit dem *Cinder* Modul realisiert. Der rohe Speicher kann als logischer Datenträger benutzt und durch das Dateisystem einer aktiven VM bereitgestellt werden. Der momentane Zustand eines derartigen *Volumes* kann auch in einem *Snapshot* festgehalten und zu einem späteren Zeitpunkt wieder eingespielt werden [Khe15]. Bei der Verwendung einer passenden Technologie, wie beispielsweise MySQL Server, sind auf *Volumes*

platzierte Datensätze als Netzwerkspeicher nutzbar. Dabei profitiert die Datenhaltung von Funktionalitäten des Ökosystems, wie der automatischen Skalierung. Die Verknüpfung eines *Block Storage* mit laufenden Instanzen kann für verschiedene Anwendungsfälle von Nutzen sein. Da der Inhalt einer VM nicht in den administrativen Problembereich von OpenStack fällt, basiert diese Funktionalität im Gegensatz zu *Glance* oder *Keystone* nicht auf Eigenentwicklungen der Community. Der Zugriff auf ein *Cinder*-basiertes *Volume* wird von unterschiedlichen konfigurierbaren *setup driver architectures* diverser Anbieter, wie IBM, NetApp, Nexenta und VMware, in automatisierter Weise ermöglicht. *Cinder* hat mehrere Technologien für die Persistierung einzelner Volumes zur Auswahl. Neben *Swift* können auch Produkte von Drittanbietern, wie beispielsweise dem verteilten Dateisystem und *Object* Datenhaltung *Gluster*, das lokale Dateisystem oder Ceph verwendet werden [KL16]. Die Bereitstellung eines Network as a Service (NaaS) soll dem Anwender einer OpenStack Architektur ein hohes Maß an *Self Service* bei der Netzwerkdefinition geben. Dies geschieht über die *Neutron* Komponente. Anwender sind mit diesem Dienst in der Lage virtuelle Netzwerke zu definieren und VM Instanzen über diese Open vSwitch (OVS)- oder linuxbasierten *Bridge*s zu verbinden. Das in [CCCS14] aufgeführte Szenario gibt einen Überblick der beiden Technologien im Bezug auf *Multi-Tenant* Netzwerkvirtualisierung. Eine Gegenüberstellung von Testergebnissen über die Ausnutzung vorhandener Bandbreiten und dem Paketdurchsatz beider Varianten, identifizierte die *Linux Bridge* als Flaschenhals, wohingegen OVS nahezu ideales Verhalten demonstrierte. Beide Technologien sind durch virtuelle *vEther* Verbindungen mit einander kombinierbar.

Die Vergabe der Media Access Control (MAC) und IP Adressen von VMs wird durch einen *Neutron Port* bestimmt, welcher als virtueller Switch angesehen werden kann. Ebenfalls können VMs Teil eines OpenStack internen, meist auf einen Tenant isolierten, Netzwerks sein. Um einen Verbindungsaufbau einer VM mit der Außenwelt des privaten Netzwerks zu realisieren, werden durch Network Address Translation (NAT) Mechanismen eine *Floating IP* auf die interne Adresse der Instanz übersetzt. Das *neutron-server* Modul nimmt API Anfragen entgegen und leitet diese an das entsprechende *neutron plugin* bzw. dessen *agent* weiter. Das Modular Layer 2 (ML2) Plugin ist eine von OpenStack bereitgestellte Kernkomponente, die verschiedene *Layer 2* Technologien unterstütz und die Koexistenz von Produkten unterschiedlicher Anbieter regelt bzw. deren Treiber anwendet. Da OpenStack typischerweise als verteiltes System betrieben wird, nehmen die Treiberagenten Anfragen an den entsprechenden Knoten entgegen und führen die gewünschte Aktion aus. Beispielsweise sind in einem Projekt drei Knoten vorhanden. Innerhalb des *controller* Knoten wird der *Neutron Server* bereitgestellt, welcher mit dem ML2 Plugin verdratet ist. Bei Anfragen an die API werden darin enthaltene Informationen an den *OVS Agent* des *compute* Knoten der Infrastruktur gesendet. Dieser konfiguriert den OVS Switch mit den gegebenen

Netzwerkdetails. Die auf diesem Knoten liegenden und am virtuellen Switch angeschlossenen VMs sind in einem weiteren Schritt mit der neuen Konfiguration entsprechenden IP Adressen und Netzwerkmasken zu versehen, weswegen der Dynamic Host Configuration Protocol (DHCP) *Agent* der *Network Node* angesprochen wird. Der Einsatz einer *Queue* ermöglicht den Datenaustausch zwischen diesen, die Anfrage bearbeitenden, *Neutron* Modulen, die auch VPNs oder eine Verbindung zu Technologien von Drittanbietern erlauben. Da sich Anbieter von physischen Netzwerkschnittstellen in Technologiedetails unterscheiden, existieren jeweils darauf zugeschnittene Python Module, die Treiberfunktionalitäten verkörpern und austauschbar sind.

Für die Umsetzung administrativer Aufgaben einer OpenStack Architektur ist die *Nova* Komponente verantwortlich. Mittels der REpresentational State Transfer (REST) Pfade eines textitnova-api Moduls werden üblicherweise Aktionen in den betreffenden Komponenten ausgeführt. Das Aufgabengebiet der *nova-compute* Komponente bezieht sich auf das Erstellen und Entfernen von VM Instanzen. Dabei wird die API des jeweiligen Hypervisors (XenAPI, Libvirt KVM oder VMwareAPI) direkt angesprochen. Des Weiteren wird durch *nova-volume* mit *Cinder* kommuniziert und *nova-network* zur Verarbeitung von *Neutron* Konfigurationen benutzt. Für die Bearbeitung eines Aufgabenmusters, welches an die *nova-api* gesendet wurde, wird dieses in einzelne Aktionen der entsprechenden Verbindungsmodule von *Nova* übersetzt und ausgeführt. Die *nova-scheduler* Komponente bestimmt dabei, auf Grundlage von in einer *Message Queue* abgelegten Informationen der einzelnen *deamons*, den Ort der auszuführenden Aktion. Eine Analyse der momentanen Auslastung von Ressourcen unterstützt den Scheduler beispielsweise in seiner Entscheidung über die Zuweisung von Aufgaben an einen bestimmen *Compute Host*.

Die grafische Benutzeroberfläche ist durch das *Horizon* Dashboard realisiert. Außer den Attributen der momentanen *session*s persistiert *Horizon* keine weiteren Informationen. Die auf dem *Django* Framework basierende Python Anwendung gibt dem Anwender die Möglichkeit der initialen Aufgabenverteilung an APIs der einzelnen OpenStack Module und deren *daemons*. Diese zustands- und datenlose Web Anwendung verknüpft Abhängigkeiten von Modulen des OpenStack Ökosystems, ermöglicht deren Bedienung und ist mit Sichten auf eigenentwickelte Dienste erweiterbar [FFG+14]. Neben dem 'branding' dieser Schnittstelle mittels eigener Darstellungen der Infrastruktur sind auch Einschränkungen von möglichen Aktionen über diese implementierbar. Ist eine problemspezifische Funktionalität mit der Ausführung mehrerer OpenStack Module realisierbar, kann dessen Verarbeitung in *Django* implementiert und hinter einer *Button*-Darstellung oder Eingabemaske in *Horizon* abstrahiert werden. Die Kausalität von Methoden, wie die Erstellung einer VM von einem vorab registrierten *Image* und festgelegten Hardwareressourcen, deren Zuweisung zu einem Netzwerk und der anschließenden Neuvergabe von Adressen durch den entsprechenden DHCP Server,

sind mit der API des jeweiligen OpenStack Dienstes in der betreffenden Python Implementation der *Panel*-Logik umzusetzen [AGS15].

Jedes CMS benötigt eine Komponente zur Überwachung der unterliegenden Infrastruktur. Der durch das *Ceilometer* Modul bereitgestellte *Telemetry* Dienst nimmt sich dieser Aufgabe an und verwendet die aktuellen Werte von Ressourcen als Grundlage für die auf den Kunden zukommenden *pay per use* Abrechnung. Der Zugang zu gesammelten Werten über Ressourcen und Instanzen wird durch den *ceilometer-api* Prozess bereitgestellt. Über diese Schnittstellen lassen sich *Meter* definieren, die sich auf eine Messung von beispielsweise der CPU-Auslastung pro Instanz bezieht. Dafür notwendige Beispielsituationen sind in der Datenbank abgelegt und repräsentieren die Attribute des jeweiligen *Meter*s. Ein auf jedem *compute node* platzierter *ceilometer-agent-compute* ist mit der Sammlung von Informationen über VM Instanzen betraut und kommuniziert diese über eine *Message Queue*. Diese Softwarekomponente wird für die Beweissicherung und Kommunikation erhobener Daten im Bezug auf das vorab formulierte *Meter* betrieben. Dabei wird eine Art der *Statistics* des *Meter*s wie `min`, `max`, `avg`, `sum` oder `count` deklariert, und durch den Agent auf gesammelten Datensätzen ausgeführt. Die Interpretation gesammelter Datensätze und deren Übersetzung in menschlich lesbare Form geschieht durch Transformatoren. Abhängig von dem Wert einer erstellten *metering message*, können auch automatische Aktionen ausgeführt werden. Diese sind in einem *Alarm* definiert und basieren auf Schwellwerte bestimmter Attribute einer durch die *Message Queue* empfangene Nachricht. Ein *ceilometer-collector* Prozess ist für die Entgegennahme von Informationen aus der *Message Queue* und deren Persistierung in einer Datenbank verantwortlich. Der *controller* Knoten einer OpenStack Infrastruktur enthält einen *ceilometer-agent-central*, der Statistiken über die Last von Ressourcen sammelt.

Mit dem Orchestrierungsdienst *Heat* lassen sich Dienste in automatisierter Weise verwalten. Die Definition von den in JavaScript Object Notation (JSON) formatierten Heat Orchestration Template (HOT) *Templates* ist die Grundlage der Verwaltung von Diensten zugehörigen Ressourcen. Die enthaltene *parameters* Liste besteht dabei aus verschiedenen Typen eingehender Werte aktiver Ressourcen, wie beispielsweise der Netzwerk- oder VM IDs eines spezifischen Benutzers. Das *resources* Attribut dieser *YAML* basierten Struktur bezieht sich auf die auszuführende Aktion im Bezug auf die deklarierten *parameters* und ist feingranular formulierbar. Beispielsweise ist ein Hypertext Transfer Protocol (HTTP) *Load Balancing* Regelsatz unter Angabe spezifischer *parameters* für die Erstellung eines hochverfügbaren Dienstes denkbar. Unter Angabe von verschiedenen Attribute über die zu instanziierenden VMs, wie *image*, dem passenden *flavor* und der *subnet_id*, lassen sich diese mit der in den *resources* definierten Strukturen in ein Cluster, in dem eine virtuelle IP Adresse von allen Instanzen geteilt wird, zusammenfassen. Mit der Bereitstellung mehrerer HOTs in der

Heat Engine lassen sich auch größere Infrastrukturen durch reine Formulierung von Regeln anwenden. Die *Heat Engine* lässt sich, wie in Abbildung 14 aufgezeigt, mit der *Ceilometer* Plattform verbinden. Diese Kombination ermöglicht bei entsprechen-

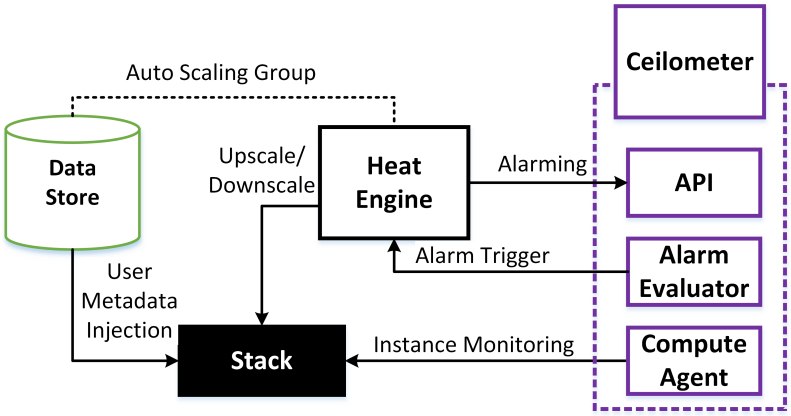

Abbildung 14.: Automatische Skalierung von Ressourcen mit Heat und Ceilometer - Eigene Darstellung in Anlehnung an [Khe15]

der Konfiguration den Betrieb automatisch skalierter Infrastrukturen. Am Beispiel der Skalierung eines Dienstes durch das Hinzufügen und Entfernen von VMs, ist eine *Heat ScalingPolicy* innerhalb der *resources* definiert und bezieht sich dabei auf die deklarieren *parameters*. Die automatische Anwendung dieser, die Anzahl betriebener VMs beeinflussenden, Regelwerke geschieht mit dem durch das *Compute Agent* Modul ausgeführte *Instance Monitoring* von *Ceilometer*. Bei der Übersteigung eines Schwellwerts kann ein *Alarm* mit der jeweiligen *Scaling Policy* einer HOTkombiniert werden. Ist die Rechenlast der Maschinen eines Cluster über einen längeren Zeitraum überdurchschnittlich hoch, wird der durch den *Alarm Evaluator* gefundene *Alarm* von der *Heat Engine* in die Instanziierung neuer VMs übersetzt. Bei wieder frei werdenden Ressourcen kann ein weiteres, durch einen *Alarm* ausgeführtes, Regelwerk zur automatischen Reduzierung der VMs eines Clusters beitragen [Khe15].

Entwurf der Datenhaltung einer OpenStack Infrastruktur

Die horizontale Skalierbarkeit von OpenStack ist einer der Erfolgsfaktoren dieser Technologie. Eine Orchestrierung beteiligter Teilsysteme kann mit dem Konzept eines *cloud controller* als logische Einheit zusammengefasst werden[Khe15]. Die den *cloud controller* beherbergenden Maschinen sind der zentrale Zugriffspunkt auf Datenbanken, Message Queues, Authentifizierung und Autorisierung, Image- und Schedulingdienste, API Endpunkte und das Web-basierte Frontend von OpenStack. Typischerweise sind sich ähnelnde Dienste wie Proxys zu Datenbanken und Message Queues oder die Frontend API und Scheduler Daemons auf einem Knoten zusammengefasst und werden zur Steigerung der Verfügbarkeit redundant vorgehalten [FFG+14]. Mit

der Verwendung eigener Backend Datenhaltungen der voneinander abhängigen *Nova, Glance* und *Cinder* Module einer OpenStack Architektur ergeben sich jedoch zwangsläufig netzwerkbedingte Latenzen und die damit einhergehende Verschwendung von Ressourcen [KL16]. Daher wird im folgenden die in OpenStack standardmäßig konfigurierte Datenhaltung einer Anwendung der Ceph Technologie gegenübergestellt.

Abbildung 15 behandelt dabei auch Strategien zur Definition von Images und deren Instanziierung. Ceph basiert auf dem Reliable Autonomic Distributed Object Store (RADOS), das für die Verteilung und Replikation von Objekten in einem Cluster verantwortlich ist und mit der *libdbd* Bibliothek Daten auf Objektebene zugreifbar macht [Khe15]. Die Verwaltung von RADOS-Objekten wird durch eine Blockspeicherschicht realisiert, innerhalb derer RADOS Block Device (RBD)s persistiert sind. Die orange gestrichelten *Placement Groups* stellen die Grundlage der Deklaration von Objektreplikationsstrategien eines Ceph Clusters dar. Eine *Placement Group* kann mehrere, physikalisch getrennte, Object Storage Devices (OSD) beinhalten, die auch einem Ordner im Dateisystem einer Maschine entsprechen können. Die Überwachung der Konsistenz dieser OSDs wird von einer *Monitoring* Komponente übernommen. Die Verteilung der Objekte in OSDs basiert auf sogenannten *CRUSH Maps*, die mittels einem Ceph-*monitor daemon server* und *metadata server* konfigurierbar sind. Das verteilte Ceph Dateisystem ermöglicht auf Grund unterschiedlicher Typen von Datenhaltungsschnittstellen zu Objekt- und Blockspeichern eine feingranularere Kontrolle über die Verteilung und Replikationsstrategien, wie die separate Datenhaltung einzelner OpenStack Module [FFG+14]. Die gemeinsame Sicht von *Nova, Cinder* und *Glance* auf Objekt- und Blockspeicher, verringert den Kommunikationsbedarf und Datenaustausch bei kollektiv ausgeführten Aktionen immens. Mit der Standardkonfiguration mögliche Szenarien im Bezug auf die sinnvolle Verteilung der Datenhaltung wurden in [FFG+14] identifiziert. Je nach Problemstellung und für die Module verfügbare Hardware ergeben sich unterschiedliche Deploymentszenarien. Alle *Glance*-verwandten Dienste können zusammen mit dem Proxy-Server zu *Swift* auf einem Knoten betrieben werden. Die physische Trennung der Metadaten der Objekte von deren tatsächlichen Datenhaltung im *Swift* Backend verlangt eine entsprechende Netzwerkverbindung zwischen beiden Knoten. Wird der *Swift* Proxy auf einer eigenständigen physischen Maschine ausgeführt, bleibt der Hauptteil der verfügbaren Ressourcen ungenutzt und rechtfertigt sich daher nur durch Overprovisioning in einer virtualisierten Umgebung. Werden alle Datenbanken auf einem zentralen Server ausgeführt, verringert sich zwangsläufig dessen Durchsatz. Die Konzentration der Datenhaltung auf einer Maschine vereinfacht jedoch durch Replikationsmechanismen auch Failoverszenarien. Bei Verwendung eines Hardware *Load Balancer* und hohem Datenaufkommen empfiehlt sich der Einsatz mehrerer *nova-api* und *swift-proxy* Prozessen auf unterschiedlichen physischen Maschinen.

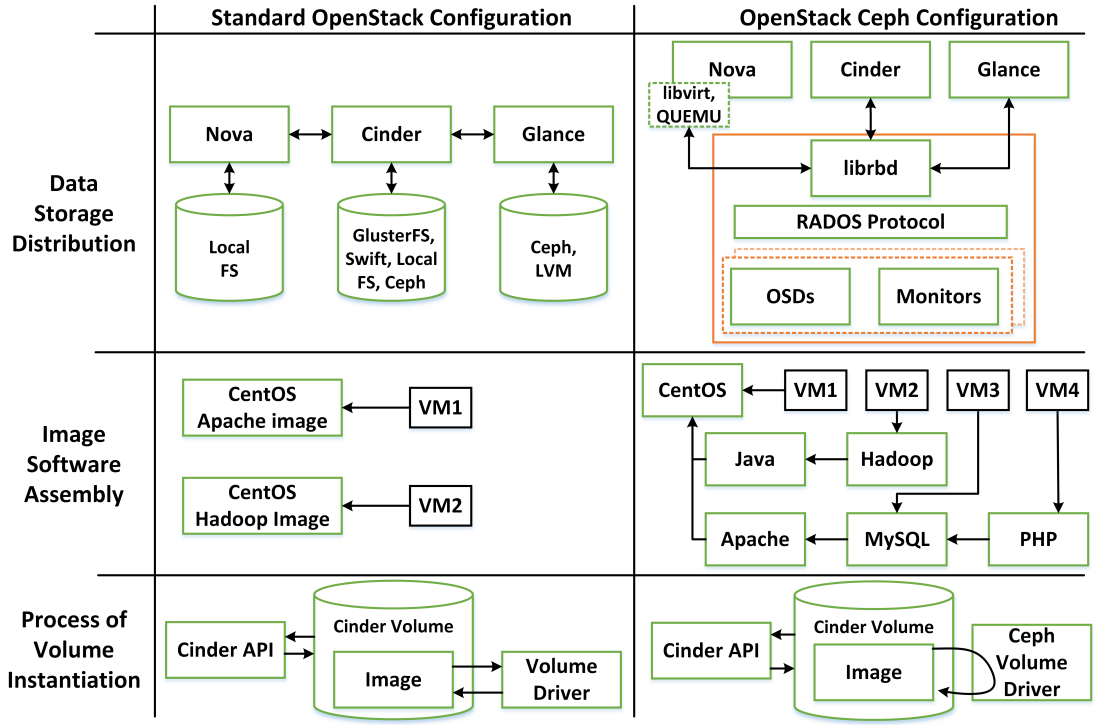

Abbildung 15.: Gegenüberstellung der standardmäßigen Datenhaltung und Ceph in OpenStack - Eigene Darstellung in Anlehnung an [KL16, Khe15, FFG+14]

Die Registrierung mehrerer vordefinierter VM-Images und *Volume Backups* ist die traditionelle Herangehensweise an die Bereitstellung unterschiedlicher, mit an eine Aufgabe angepasster Software bestückter, Plattformen. Bei Vorhalten von auf einander aufbauenden Imageversionen wird dementsprechend viel Speicherplatz verbraucht. Das Konzept dieser *Single-level Images* ist, verglichen mit dem in Ceph angewandten *copy-on-write* Mechanismus, unwirtschaftlich. Mit dieser als *thin provisioning* bezeichneten Methode ist die Abbildung einer Imagehierarchie realisierbar. Ein sogenannter *clone* beinhaltet, ausgehend von dem *Base Image*[2], alle individuell angereicherten Softwarestände und eliminiert daher die in einem *Single Level* existierenden duplikate [KL16]. Die auf ihren Vorgänger aufbauenden *Clone Images* verbrauchen daher nur den seit dem letzten *copy-on-write* Zustand zusätzlich benötigten Speicherplatz bei der Erstellung eines neuen Levels. Durch Anbieten jeden Levels der Softwarekette einer komplexen Plattform als instanziierbares Image, wird dem Endanwender eine breitere Auswahl an vordefinierten Grundstrukturen durch einen einzelnen implementierten Anwendungsfall gegeben.

Die *copy-on-write* Eigenschaften kommen auch der raschen Bereitstellung der von einem *Clone* instanziierten Ressourcen zu gute. In beiden Fällen der letzten Zeile aus Abbildung 15 wird eine Instanziierungsanfrage eines durch *Glance* verwalteten Images

[2]Das Base Image in der mittleren Zeile aus Abbildung 15 ist das grundlegende Betriebssystem *CentOS*.

an der *Cinder API* entgegengenommen. In der Standardvariante muss das jeweilige Image auf die *compute node* übertragen werden, wohingegen die in Ceph bei der Importierung von Volumes genommenen Snaphots als grundlegende *Clone Images* betrachtet werden [FFG+14].

Standortübergreifende OpenStack Infrastrukturen

Die hohe Nachfrage nach einem angebotenen Dienst führt in vielen Fällen zur Anschaffung neuer Hardware und deren Konfiguration in bestehende Infrastrukturen. Bei der Erschaffung neuer Außenstellen oder Filialen eines auf OpenStack basierenden Unternehmens, sind deren VM Instanzen in das bestehende Firmennetzwerk integrierbar. Eine *Neutron* Erweiterung stellt VPN as a Service (VPNaaS) Funktionalitäten zur Verfügung, durch die Kommunikation zwischen privaten OpenStack Netzwerken mit virtuellen Routern über Internet Protocol Security (IPSec) getunnelt werden können [Khe15]. Mit Aufsetzen einer *IPSec site connection* werden die in einer Internet Key Exchange (IKE) Policy definierten Authorisierungs- und Verschlüsselungsalgorithmen der in einer IPSec Policy definierte *Encapsulation Mode* Modus der Kommunikation verwendet. Alle involvierten CMS Systeme müssen sich auf ein gemeinsames Subnet einigen und die jeweiligen *remote peer subnet*s samt Zugangsschlüssel zum VPN angeben. VM Instanzen eines Tenant können nachfolgend dem jeweiligen Subnet zugewiesen werden.

4.3. Web Application Frameworks

Die Bereitstellung einer netzwerkbasierten Applikation beinhaltet meist statische Teile wie Hypertext Markup Language (HTML) Templates oder die darauf zeigende URL. Um Informationen der dynamischen und individuell ausformulierten Programmlogik auf das *Application Layer* des Open Systems Interconnection (OSI) Modells zu übertragen, bedarf es einer entsprechenden Technologie bzw. Rahmenwerk für die Bereitstellung netzwerkbasierter Kommunikation. Die jeweilige Laufzeitumgebung einer Web Anwendung hat in den meisten Fällen Einfluss auf nicht-funktionale Eigenschaften wie Sicherheit, Skalierbarkeit oder Wartbarkeit der darin ausgeführten Logik. Derartige Qualitätseinbußen finden sich häufig in Web-basierten Applikationen, die in PHP, Java, oder dotNET verfasst wurden [Geo16]. Im Folgenden werden zwei alternative Rahmenwerke vorgestellt, deren Stärken sich durch die Architekturen der Laufzeitumgebung und Möglichkeiten bei der Verwendung verschiedener Bibliotheken auszeichnen.

Webanwendungen mit Python-basiertem Django Framework

Die Ausführung von formulierten Anwendungen in der GPL Python basiert auf der Interpretation von Skripten. Diese Hochsprache bedient neben der objektorientierten Programmierung weitere Paradigmen, wie imperative, prozeduale, reflektive und funktionale Implementierungsmuster [Sha13, Pil09]

Python Anwendungen sind auf Grund von für diverse Betriebssysteme kompilierten Interpretern nahezu so unabhängig von ihrer Umgebung wie in einer Java Virtual Machine (JVM) ausgeführte Prozesse. Die virtuelle Laufzeitumgebung einer VM beinhaltet prinzipiell eine Menge an atomaren Anweisungen, die unabhängig von einer Sprache sind und meist als isolierte *Sandbox* betrachtet werden. Die Ausführung der von einander unabhängigen Anweisungen oder entsprechend kompiliertem *Byte Code* ist deterministisch und bis auf den momentanen Zustand der VM, bzw. deren Einwilligung in die Operation, auf keine externen Informationen angewiesen. Ein Interpreter analysiert hingegen die Syntax einer spezifischen Sprache, die aus nicht isolierter Symbolik besteht. Die auf einander folgenden Befehle der textuellen Quelle sind auf Grund von Relationen zu anderen Programmteilen und den einen Anweisungsblock umfassenden Strukturen in den meisten Fällen nicht atomar. Eine im Code deklarierte Aktion wird daher während der Abarbeitung als eingehender Parameter für den Interpretationsprozess benutzt. Der Komfort von ausführbarem und gleichzeitig lesbarem Quellcode kommt wegen der Interpretation von komplexen Sprachen und verschachtelten Anweisungen zum Preis von Performanzeinbußen während der Ausführung.

Das Python-basierte *Django* Framework ermöglicht ein effizient skallierbares, wartbares und sicheres Entwickeln von Netzwerkanwendungen [Geo16]. Die interaktive Python *Shell* kann hierbei für die Administration einer Applikation angewandt werden. Nicht zuletzt wegen dem Zusammensetzen vordefinierter Strukturen bzw. Muster und der damit verringerten aufzubringenden Entwicklungszeit eines komplexen Systems wird dieses Rahmenwerk im Folgenden genauer betrachtet.

Die Grundstruktur einer *Django* Anwendung kann bei entsprechend konfigurierten Umgebungsvariablen durch `django-admin startproject [project name]` generiert werden. Das innerhalb des `[project name]` platzierte *manage.py* Modul ist für die weitere Verwaltung des Projekts verantwortlich. Ein *Django* Projekt wird daher im folgenden durch `python manage.py startapp [application name]` mit beliebig vielen Applikationen angereichert. Zusätzlich kann ein `-template [template name]` Parameter als weitere Spezifizierung der Struktur einer Applikation mit angegeben werden und den Pfad zu einer Template Datei, URL oder kompressiertem Archiv repräsentieren.

Die generierte Anwendung basiert auf einer eigenen Datenhaltung, die einzelne *Models* beinhaltet. Das Schema einer Applikation ist in deren *models.py* definiert und nach einer Änderung durch Ausführung von `python manage.py migrate` auf Projektebene mit der Datenbank zu synchronisieren. Die in der *views.py* abstrahier-

te Programmlogik einer Anwendung kann als REST Schnittstelle oder dynamische HTML Repräsentation implementiert werden. Eine *urls.py* bestimmt den Pfad zu diesen Sichten und leitet die Anfrage gegebenenfalls auf die Darstellung in einem auf resultierenden Informationen angepassten HTML Template um.

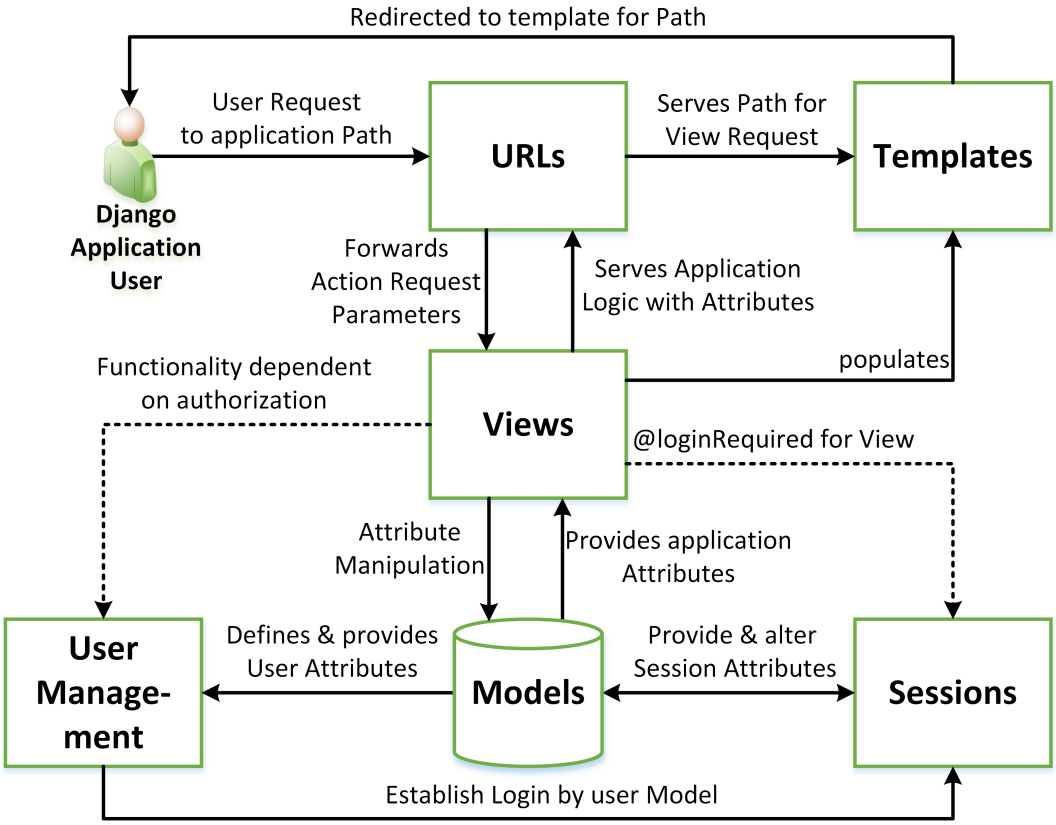

Abbildung 16.: Komponenten eines Typischen Django Anwendungsfall

Abbildung 16 zeigt alle in einen typischen Anwendungsfall involvierten Komponenten eines *Django* Systems auf. Für den Benutzer einer *Django* Anwendung werden alle modularen Abhängigkeiten hinter einer URL abstrahiert. Unter Angabe eines auf die Web Anwendung zeigenden Pfades wird dem *Django Application User* die implementierte Sichtweise auf das entsprechende HTML-*Template* zurückgegeben. Die Anfrage an einen solchen Pfad kann ebenfalls einer JSON Struktur resultieren. Programmtechnisch wird die Bereitstellung der Logik innerhalb des *urls.py* Moduls mit der Deklaration einzelner, auf die jeweilige Implementation verweisenden Pfade, realisiert. Die in einer *views.py* definierten Sichten beinhalten dabei die dem Pfad zugrundeliegende Anwendungslogik und befüllen das entsprechende Template. Das *User Management* des Projekts basiert, sofern nicht die standartmäßige Konfiguration genutzt wird, auf Benutzerprofilen, welche mit projektspezifischen Attributen angereichert sind. Je nach Implementierung sind bestimmte Sichten nur in einer bestimmten Rolle anwendbar und verlangt eine Authentifizierung des Benutzers. In Modellen ausgedrückte Objekte der Datenbank können ebenfalls für die Bearbeitung einer Aufgabe manipuliert

oder zur Darstellung in einem HTML-*Template* angefragt werden. Die spezifizierte Problemstellung beinhaltet meist auch auszuführende Aktionen, mit denen die Anwendung steuerbar gemacht werden soll und letztendlich Teil der API des Projekts wird. Eine Anwendung kann durch die Verwendung von ausgewählten *Django* Strukturen und Komponenten auf die Funktionalitäten der jeweiligen Prolemstellung angepasst werden [EL14].

Web Anwendungen durch funktionale Programmiermodelle

Erlang ist eine funktionale Programmiersprache, deren Grundgedanke auf Stabilität, Ausfallsicherheit und der Gewährleistung massiver Parallelität beruht. Diese wurde von *Ericsson Computer Science Laboratory* im Jahre 1986 entwickelt und ist seither im Bereich der Telekommunikation vertreten. Die Verwendung eigens bereitgestellter Laufzeitumgebungen zur Ausführung von Applikationen ist eine verbreitete Herangehensweise bei der Generalisierung von Hochsprachen in unterschiedlichen Systemarchitekturen. Das Konzept der Bjorn Gustavsson/ Bogumil Hausman Erlang Abstract Machine (BEAM) basiert zwar auf einem einzelnen Systemprozess der die VM repräsentiert, jedoch sind in dieser typischerweise hunderte oder gar tausende leichtgewichtige Erlang-Prozesse aktiv. Grundlegend besteht ein Erlang Projekt aus verschiedenen Modulen, die dem Event-Loop Muster folgen und unter Anwendung des Aktorenmodells kommunizieren. Deren Lebenszyklus und Arbeitsweise ist strikt voneinander isoliert. Ein weiteres Prinzip von Erlang ist das dynamische Starten von Modulen zur Laufzeit. Betrachtet man die Arbeitsweise einer aus mehreren Modulen bestehenden Funktionalität der Anwendung, lassen sich oft hierarchische Muster identifizieren. Wird eine von einander abhängige Prozesskette erst beim Auftreten einer bestimmten Situation erstellt, verringert sich die durch den Betrieb der BEAM verursachte Grundrechenlast. Alle der BEAM bekannten Module können ebenfalls durch direkte Interaktion mit der Konsole dynamisch in den Applikationslebenszyklus einbezogen werden. Der Rückgabewert einer mit dem betreffenden Erlang-Modul und problemspezifischen Attributen parametrisierten *spawn* Funktion besteht aus einem Tupel, welches die Erlang Process ID (PID) beinhaltet. Jeder Erlang Prozess besitzt eine mit der jeweiligen PID assoziierte Mailbox, in der Nachrichten von anderen Prozessen entgegen genommen werden. Die Methodik dieses asynchronen Kommunikationsmechanismus ist nach [Heb13] wie folgt beschrieben: *"In a nutshell, if you were an actor in Erlang's world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message"*. Diese Umstände begünstigen Millionen parallel ausgeführte Prozesse, die in Abhängigkeit voneinander Interagieren. Die Fehlertoleranz bei der Umsetzung einer Ereigniskette ist neben dem implementierten Anwendungscode auch auf höherer Ebene zu betrachten. Diese leichtgewichtigen Prozesse stehen unter der Aufsicht von

sog. *Supervisor* Komponenten, die auftretende Probleme einzelner BEAM Prozesse erkennen und gegebenenfalls neu starten[CT09].

Eine Integration von OS-Prozessen in Erlang Anwendungen ist ebenfalls realisierbar. Die in beliebigen Programmiersprachen verfassten und bereits durch den zugehörigen Compiler übersetzten, Programme werden durch Erlang gestartet und in die Supervisor-Hierarchie eingebunden [Wol15]. Der Nachrichtenaustausch zwischen Erlang-Prozessen und den in fremden Laufzeitumgebungen platzierten Anwendungen ist dabei jedoch nicht definiert und muss manuell, mit beispielsweise *Message Queue*s oder Technologien wie *Apache Thrift*, umgesetzt werden.

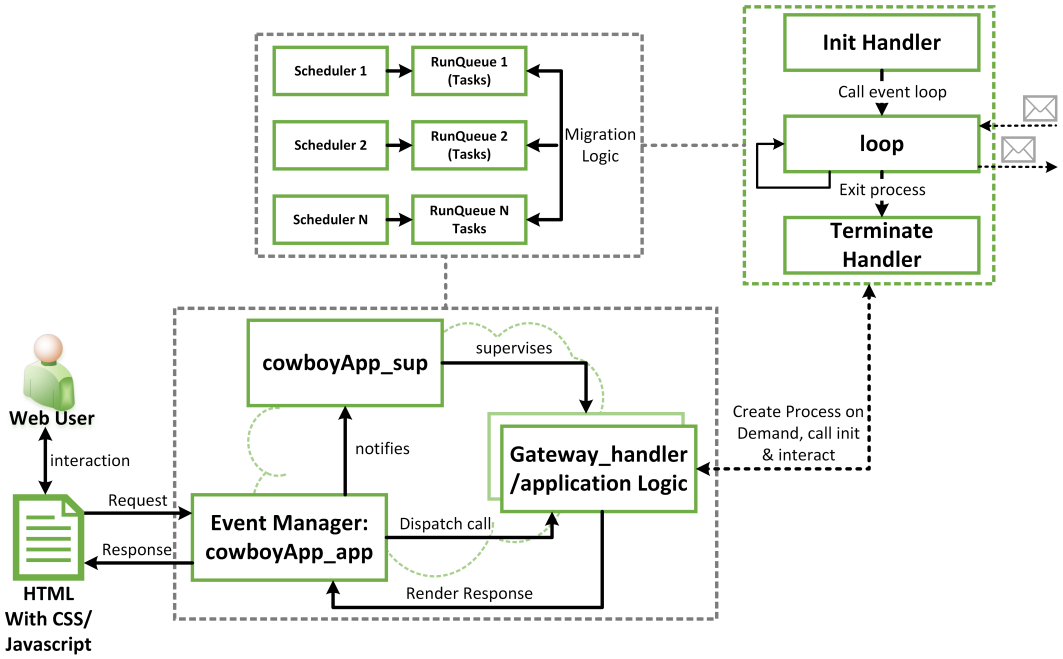

Abbildung 17.: Erlang Web Anwendung mit dem Cowboy Framework

Abbildung 17 zeigt eine auf Erlang basierende Web-Anwendung, die den leichtgewichtigen REST Server *Cowboy*[cow16], der auf Prinzipien des Open Telecom Platform (OTP) aufbaut, als Gateway zwischen Javascript Funktionen einer HTML und den Modulen des angebotenen Dienstes verwendet. Erlang Module können ein bestimmtes Verhalten durch Deklaration des *-behaviour(..)* Parameter annehmen. Das Event Managermodul stellt die jeweiligen REST calls zu Verfügung und agiert daher als dauerhaft aktive *application*. Eingehende Anfragen an eine implementierte REST Schnittstelle werden durch diese Komponente bereitgestellt und mit den jeweiligen *Gateway Handler* verknüpft. Die Verhaltensdefinition eines Moduls als *cowboy_http_handler* impliziert die Implementierung der *handle* Funktion, in der eine den Pfad betreffende Aufgabe gestartet wird. Die Antwort an das aufrufende Router-Modul kann durch Cowboy Funktionsaufrufe von Erlang Attributen in das letztendlich an den Benutzer übertragene JSON Objekt transformiert werden. Bis diese Nachricht in der HTML Repräsentation dargestellt wird, können jegliche der BEAM bekannten

Erlangmodule parametrisiert gestartet werden.

Die Ausführung mehrerer BEAMs innerhalb einer physischen oder virtualisierten Hardware steigert die Skalierbarkeit einer Erlang Applikation nicht nur durch den Einsatz von Multicore Prozessoren. Jede BEAM beinhaltet einen eigenen Scheduler, der eine Menge an aktiven Prozessen bearbeitet. Da alle *RunQueue*s über eine Migrationslogik mit einander verknüpft sind, kann eine gleichmäßige Auslastung aller CPUs erreicht werden. Nebenläufige Prozesse, die auf physischen Datenbestand zugreifen, müssen sich untereinander dennoch koordinieren. Die Verteilung der Laufzeitumgebung in einem Netzwerk ist nicht ausgeschlossen. Eine *Erlang Node* kann sich über beliebig viele CPU Kerne eines Hosts erstrecken, muss in einem verteilten System jedoch eindeutig identifizierbar sein. Unter zusätzlicher Angabe von IP Adressen oder DNS Namen eines Knotens, leitet dieser empfangene Nachrichten an bestehende Prozesse des eigenen Gültigkeitsbereichs weiter. Die Erstellung von problemspezifischen Prozessen eines Moduls auf mit spezieller Hardware ausgerüsteten Maschinen, kann somit auch als entfernter Funktionsaufruf im Programmcode formuliert werden.

Mit Buildsystemen wie beispielsweise *erlang.mk* ist das Grundgerüst einer Anwendung generierbar. Entstehende *app* und *sup* Ordner sind mit Modulen für die Bereitstellung der Applikationslogik und dem entsprechenden Supervisor zu füllen. In einem weiteren Schritt ist die ebenfalls aus der *erlang.mk* resultierende *Makefile* mit allen für das Projekt ausgewählte Abhängigkeiten anzureichern und auszuführen. Mit der Definition von *Git Repositories* und deren unterschiedlichen Versionen oder *Branches* ist auch öffentlicher oder organisationsinterner Code in die zu erstellende Applikation integrierbar.

Nach Anreicherung und Ausführung des *Makefile* resultiert das Projekt in einem *Release*, welches das System starten kann und als Grundlage des *Code Replacement*s zur Laufzeit benutzt wird. Bei der Aktualisierung eines Moduls sind alle auf einer früheren Version des Projekts basierenden Module, die modifiziert wurden, in diesem Paket innerhalb eines die Version repräsentierenden Unterordners festgehalten. Im jeweiligen Zielsystem kann das Release nach einer Überprüfung der Syntax aller Komponenten durch die BEAM installiert werden.

Existieren zyklische Abhängigkeiten zwischen einzelnen Modulen des Projekts, verringert sich die Flexibilität des *Deployment* einer neuen Version während dem Betrieb der betreffenden Gesamtanwendung. In einem solchen Fall ist ein Neustart der gesamten Erlang Applikation meist unumgänglich [Heb13]. Durch die Kombination des Supervisor Prozesses mit dem asynchronen Nachrichtenversand einzelner Prozesse an die *Mailbox* anderer, kann ein *'let it crash'* Konzept umgesetzt werden, da bei fehlerhaftem Verhalten beeinträchtigte Komponenten neu konfiguriert und gestartet werden. Bei sauberer Implementierung und Definition von *Supervisor*-überwachten Prozessbäumen sind Szenarien, bei denen der komplette Service beeinträchtigt wird,

vermeidbar. Daher ist trotz der gemeinsamen Ausführung innerhalb eines Prozesses eine vollständige Isolation gegeben [Wol15].

4.4. Definition individueller Cloud Images

Ein Cloud Image repräsentiert eine Menge an daraus instanziierten VMs und besteht prinzipiell aus einem auf den Einsatz in einem Netzwerk ausgelegten Betriebssystem. Die Anreicherung dieses Images mit Programmen, Standartbenutzern oder der automatischen Registrierung mit einem unternehmensinternen Rechtesystem verringern die letztendlich benötigte Bearbeitungszeit einer Bereitstellung und damit möglicherweise auch die Auslastung von Mitarbeitern. Der Einsatz maßgeschneiderter Cloud Images wird sowohl in Kapitel 5 als auch in den in Kapitel 6 gezeigten Anwendungsfällen wieder aufgegriffen.

Die *libguestfs* Bibliothek beinhaltet Werkzeuge für den Zugriff und die Modifizierung von VM Images im *Raw* oder *qcow2* Format. Eine *python-guestfs* Bibliothek ermöglicht die Anwendung dieses Werkzeugs in Python Modulen und kann daher auch innerhalb von Anwendungen als Problemlösung verwendet werden [CR15].

Mit dem *virt-builder* Kommandozeilenwerkzeug lassen sich neue Images anhand von Minimalversionen verschiedener Betriebssysteme, die das System während der Ausführung eines Befehls aus einem Repository herunterlädt, erstellen. Neben Angaben über das Format des betreffenden Images und der Architektur dessen grundlegenden Betriebssystems, wird die Mächtigkeit dieses Werkzeugs durch die Kombination mehrerer Parameter ersichtlich. Mit der Anwendung beliebiger Befehle durch den *run-command* Parameter ist ein Cloud Image prinzipiell in jeglicher Art manipulierbar, da diese Kommandos mit root-Privilegien ausgeführt werden. Da einige Anwendungsfälle auf dynamische Attribute, wie dem Standort einer VM, angewiesen sind, existiert ein Mechanismus zur Ausführung von Befehlen während dem ersten Bootvorgang eines instanziierten Cloud Image. Die Übergabe von Skripten an *virt-builder* ermöglicht die Wiederverwertung von bereits bestehenden Konfigurationsskripten zur Erstellungszeit des Cloud Image. Durch die Anwendung des jeweiligen Paketmanagers einer Distribution sind Anwendungen in das resultierende Abbild integrierbar. Ebenfalls können auf dem Hostsystem gelagerte Dateien in das Gastsystem kopiert und einer Änderung der Zugriffsrechte unterzogen werden. Die Manipulationen der im Image platzierten Dateien ist auch durch einen Parameter realisierbar. Die Erstellung von Standartbenutzern des Image kann mit einem *run-command* umgesetzt, aber im Nachhinein mit einem bereits vorhandenen und auf dem Hostsystem platzierten Secure SHell (SSH)-Key assoziiert werden. [vir17]

Die Verwendung von VMs ist im Bereich der *Malware Analysis* ein weit vertretenes Konzept zur Abschirmung riskanter Software von den Produktivsystemen einer Un-

ternehmung. Auf diese Problemstellungen bezogene Projekte, wie *MalBoxes*[3] nehmen sich der Analyse von Windows-basierten Anwendungen in virtuellen Umgebungen an [Bil17]. Dieses auf *Packer* und der *Vagrant* Technologie basierende Werkzeuge sollen die Erstellungszeit einer mit auffälligen Dateien bestückten isolierten VM verringern. In einer *Vagrant* Konfigurationsdatei deklarierbare Parameter, wie Zugangsdaten zu VM-Instanzen oder darin zu platzierende Analyseanwendungen, sind mit der Arbeitsweise von *virt-builder* zu vergleichen. Mit dem Pfad zu einem die zu analysierenden Dateien beinhaltenden Ordner sind ebenfalls weitere Werkzeuge in die VM zu integrierbar. Neben der Beschränkung auf die *VirtualBox* Technologie bezieht sich *MalBoxes* ausschließlich auf Windowssysteme und zeichnet sich durch beim Starten der VM in der *Powershell* ausgeführte Skripte zur Isolation des Systems aus.

Wie in Abschnitt 4.2 beschrieben, ist die Verwaltung von *Cloud Image*s durch das verteilte *Ceph* Dateisystem ein eleganter Ansatz im Bezug auf OpenStack. Aufgeführte Vorteile ergeben sich jedoch nicht in Systemen, in denen kein direkter Zugriff auf die Manipulation von Abbildern oder der Instanziierung aus Snapshots besteht. Das Verhalten bei der Instanziierung einer VM aus einem Blockspeicher muss ebenfalls manuell definiert werden. In cloudbasierten Angeboten wie der Amazon EC2 kann auf Grund von fehlenden Sichtbarkeiten demnach nicht von diesen Eigenschaften gebrauch gemacht werden.

4.5. Big Data Technologien

Handelt es sich bei einer Anforderung an große Datenmengen um transaktionale Funktionalitäten, ist dieses Problem dem Online Transaction Processing (OLTP) zuzuordnen. Dabei sind in den meisten Fällen nur wenige Datensätze innerhalb einer lesenden Aktion betroffen, durch *Random Access* geschriebene Informationen steigern dabei jedoch die Zugriffszeiten. Derartige Operationen sind auf die Aktualität der dafür verwendeten Daten angewiesen und müssen vor nebenläufiger Manipulation durch eine separate *Query* geschützt werden.

Das hohe Aufkommen der unterschiedlichen Strukturen von Informationen im *Big Data* Umfeld lässt sich mit traditionellen Verfahren nur in bestimmten Gesichtspunkten verarbeiten. Das Verlangen nach Analysen im Bezug auf gigantische Datenmengen kann dennoch mit OLAP-Techniken umgesetzt werden [CML14]. Dafür relevante Informationen werden meist sequenziell gelesen und durch Aggregationen durch *Batch Processing* in eine neue Erkenntnis übersetzt [ER16]. Diese Art von *Big-Data* Verfahren wird in der Regel durch einen Unternehmensinternen Analyst durchgeführt und soll meist bei der Entscheidungshilfe einer spezifischen Fragestellung unterstützen.

Da Datenstrukturen dieser OLAP Systeme meist auf organisationsinternen Infor-

[3] `https://github.com/GoSecure/malboxes`

mationen aufbauen, können diese in einem *Data Warehouse* gesammelt werden. Ein Hauptproblem bei dieser zentralen Datenhaltung ist die Heterogenität von Informationen der unterschiedlichen OLTP Systeme eines Unternehmens [LBR16]. Diese sind meist nur schwach miteinander assoziiert, was das mit einer auf die jeweiligen Strukturen angewendete Extraktion, Transformation und Einspeisung in das *Data Warehouse* in einen gemeinsamen Kontext gebracht werden kann.

Datenhaltung

Das mit dem *Hadoop* Ökosystem bereitgestellte Hadoop Distributed File System (HDFS) kann als verteiltes Dateisystem angesehen werden [WS14]. Die durch einen *Namenode* orchestrierten *Datanode*s beherbergen unveränderliche Datenblöcke. Der Zugriff und die Manipulation von darin enthaltenen Daten wird durch Anfragen an den *Namenode* bereitgestellt, der auch für die Replikation von Blöcken in einem Cluster verantwortlich ist [WS14]. Mit diesen Eigenschaften kann sich der Entwickler eines Anwendungsszenarios auf die eigentliche Problemstellung konzentrieren, anstatt Zeit in die Mechanismen zur unterliegenden Datenhaltung zu investieren [Dat13].

Apache Cassandra ist eine hochskalierbare *No-SQL* Datenbank. Im Gegensatz zu HDFS ist das verteilte Cassandra File System (CFS) nicht von einer zentralen Managementkomponente abhängig. Vielmehr kommt dabei eine *peer-to-peer* Ringarchitektur zum Einsatz, bei der alle Knoten in der selben Art und Weise agieren. Grundlegend werden alle Daten in zwei Spaltenfammilien gehalten. Eine *inode* Spaltenfamilie beinhaltet die Metadaten der in der *sblocks* Spaltenfamilie angesiedelten Datenblöcke. Mit dieser Technologie sind durch Aufteilen von Verarbeitungs- und Analyseclustern ebenfalls performante Echtzeitanalysen möglich[Dat13].

Batchverarbeitung

Das von Google in [DG08] vorgestellte *MapReduce* Framework galt lange als *State of the Art* im Bezug auf die Verarbeitung gigantischer Datenmengen. Die zu bearbeitenden Datenstrukturen werden dabei in *Key-Value* Paare transformiert und an multiple *Map* Funktionen weitergereicht. Eine aus dieser Funktion entstehende Liste neuer Werte-Paare wird anschließend gruppiert und mehreren *Reduce* Funktionen zur parallelen Aggregation bereitgestellt. Mit dem auf Big-Data konzentrierten quelloffenen *Hadoop* Ökosystem ist diese Verarbeitungslogik in einem verteilten System auf nahezu beliebiger Hardware umsetzbar[WS14]. Der Cluster-Manager Yet Another Resource Negotiator (YARN) orchestriert die in eine Unternehmung involvierten Maschinen, welche die kollektive Verarbeitung von in HDFS gehaltenen Datenstrukturen mit der Verarbeitungslogik *MapReduce* anhand konkret implementierter Aufgaben

analysiert und einen problemspezifischen Mehrwert erzeugt. Die in diesem linear skalierbaren Ökosystem auftretenden Fehler bei der Verarbeitung von Daten kann direkt auf Anwendungsebene erkannt und behandelt werden.

Die Aufgabendefinition basiert vollständig auf der *Hadoop* Java Bibliothek und verringert den Grad der Komplexität bei der Implementierung einer durch dieses Rahmenwerk durchgeführten Aufgabe. Daher wird keinerlei manuelle Netzwerkprogrammierung verlangt. Die für die Verarbeitung genutzten Zwischenergebnisse der Datenstrukturen werden dabei auf der ausführenden Maschine gehalten, die bei einem unerwarteten Abbruch zur direkten Wiederaufnahme der Tätigkeiten verwendet werden können. Die Implementierung eines *MapReduce Job*s ist dabei für die intelligente Auslastung von Ressourcen verantwortlich, was einen fundamentalen Wissensstand über die Infrastruktur der Analyseplattform voraussetzt [WS14].

Die *Apache Spark* Technologie wurde maßgeblich von *MapReduce* beeinflusst und stellt ein alternatives Rahmenwerk für die Verarbeitung riesiger Datenmengen zur Verfügung. Dieses ebenfalls verteilte, linear skalierbare und fehlertolerante System basiert im Gegensatz zu *Hadoop* auf Resilient Distributed Dataset (RDD)s, das eine über das Cluster verteilte und partitionierte Datenhaltung ermöglicht. Ähnlich wie in HDFS sind RDD Datensätze unveränderlich, jedoch sind darin beliebige Python, Java, Skala oder eigens definierte Objektstrukturen bzw. Klassen enthalten. Ein weiterer Unterschied ist die Verarbeitung von Informationen im Hauptspeicher, was die Zugriffszeit auf bereits gelesene Datensätze verringert. Die verarbeitenden Knoten eines *Spark* Clusters nutzen außerdem einen Mechanismus zur verteilten Sicht auf den jeweils bestehenden *Cache*. Bei der Anforderung eines bereits durch einen anderen Knoten des verteilten Systems gelesenen RDDs, wird damit ein weiteres Mal der Lesezugriff auf persistenen Speicher vermieden. Die Informationen der Verteilung über mehrere Knoten ermöglicht dabei auch die Wiederherstellung von Daten im Fehlerfall auf einer aktiven Maschine des Clusters. Die letztendliche Verteilung von implementierten Aufgaben wird aus dem Anwendungscode abstrahiert und durch das Rahmenwerk zu einer möglichst performanten Verarbeitung auf einzelne Knoten verwiesen [ER16].

Auf den Elementen eines RDD kann durch den *Cluster Manager*, welcher auf YARN oder gleichartigen Implementationen aufbaut, parallel zugegriffen werden.

In Abbildung 18 ist die Verarbeitung einer iterativen *MapReduce* Aufgabe unter Anwendung von *Hadoop* und *Spark* aufgezeigt. Die in *Hadoop* unumgänglichen Lese-und Schreiboperationen von involvierten Knoten, welche eine *MapReduce* Aufgabe verarbeiten, verringern die Performanz bei iterativen Problemstellungen. Ist eine *Reduce* Phase abgeschlossen, müssen diese Ergebnisse in das HDFS geschrieben und in darauf aufbauenden Iterationen wieder gelesen werden. Innerhalb eines *Driver Program* sind RDDs durch die Parallelisierung einer bestehenden Menge an Informationen oder der Refferenzierung eines externen Datenspeichers erstellbar. Mit der Bereitstellung

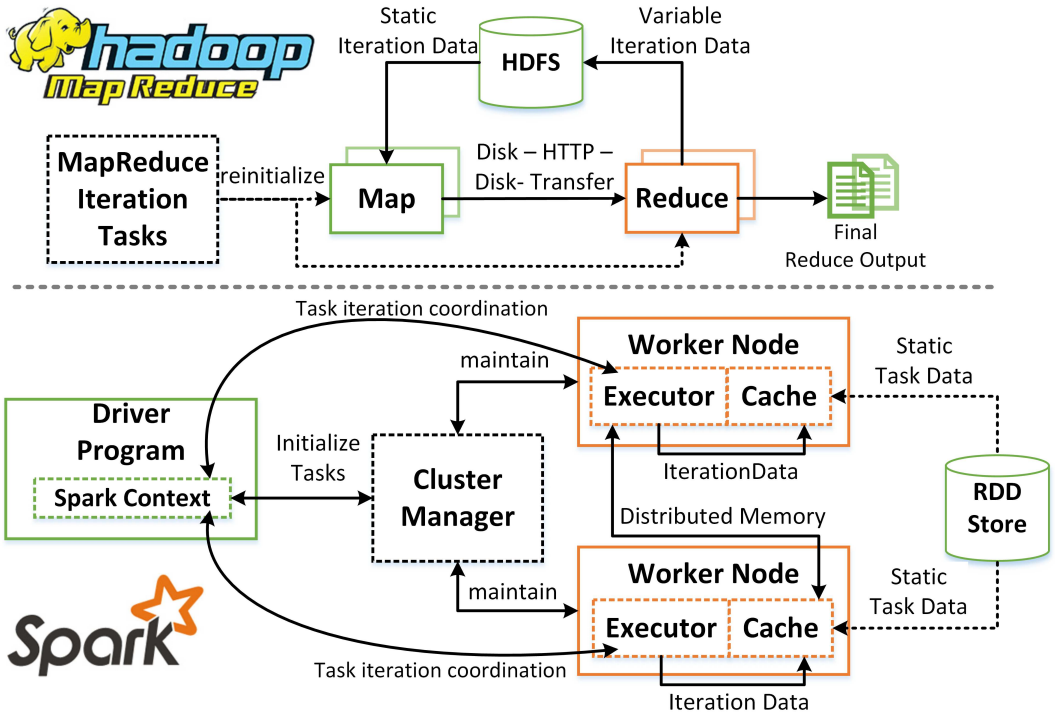

Abbildung 18.: Iterative Verarbeitung mit Hadoop MapReduce und Spark - Eigene Darstellung in Anlehnung an [ER16]

von initialen Datensätzen und berechneten Zwischenergebnissen im geteilten Hauptspeicher einer *Spark* Anwendung kann auf das wiederholte Einlesen von persistentem Speicher verzichtet werden. Verglichen mit *Hadoop*, kann die Verarbeitungszeit von Datensätzen in einer *Spark* Anwendung bis um den Faktor 100 gesenkt werden.

Verarbeitung von Datenströmen

Die Java-basierte *Apache Sorm* Technologie ermöglicht die Verarbeitung von Datenströmen in Echtzeit und basiert auf der Kommunikation von Ereignissen über mehrere zu implementierende Komponenten, die zu einer Verarbeitungskette zusammengefasst wurden. Dieses verteilte, zuverlässige und fehlertollerante System ist Open Source und unterstützt multiple Programmiersprachen. Die Skallierung der einzelnen Komponenten kann sowohl vertikal als auch horizontal bewerkstelligt werden. Mit der *at-least-one* Zustellungsgarantie von Ereignissen ist auf die Anwendung der Funktionalität von Technologien zu achten, die mit der Storm Applikation verknüpft sind. Die *exactly-one* Semantik bei der Implementierung basiert dabei auf der eindeutigen Identifizierung eines versendeten *Event*s. Wie in Abbildung 19 aufgezeigt, besteht ein *Storm* Cluster aus einem *Master Node* und mehreren *Worker Node*s. Der auf dem *Master Node* platzierte *Nimbus daemon* ist für die zentrale Überwachung und Orchestrierung der momentan ausgeführten *Storm* Anwendungen verantwortlich. Die Bearbeitung von Aufgaben geschieht in *Worker Node*s, die eine JVM bereitstellen. Darin können meh-

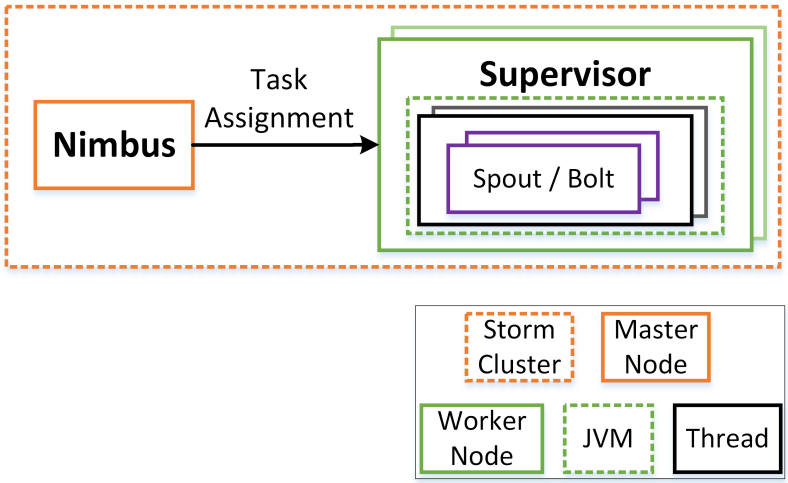

Abbildung 19.: Architektur von Apache Storm

rere Java Threads enthalten sein, welche wiederum mit dem Betrieb von *Spout*s oder *Bolts* betraut sind. Eine mit diesen Komponenten erstellte Topologie muss in jedem Fall in einem gerichteten azyklischen Graphen abbildbar sein, da sonst eine mehrfache Verarbeitung desselben Ursprungsereignisses nicht ausgeschlossen werden kann.

Die *Spout*s stellen den Einstiegspunkt einer *Storm* Topologie dar und sind meist auf die Entgegennahme und Filterung von Datenströmen einer spezifischen Quelle abgestimmt. Die Verarbeitung von gesammelten Informationen wird ausgehend von diesen Topologiekomponenten an eine Kette an verarbeitenden *Bolt*s weitergereicht. Diese Hauptkomponenten einer *Storm* Applikation kommunizieren über sogenannte *Stream*s, die eine Verbindung von *Spout* zu *Bolt* und *Bolt* zu *Bolt* repräsentieren. Die *Storm*-internen Datenströme bestehen dabei aus immer aus einer Liste an eine ungebundene Anzahl geordneter Elementen (Tupel), die durch einen vergebenen Namen abstrahiert ansprechbar sind. Die Elemente eines *Streams* können primitive Datentypen oder durch die Formulierung eines *Serializer*s auch komplexe Strukturen beinhalten.

Der *Nimbus daemon* eines sich über mehrere Maschinen erstreckenden *Storm* Clusters erkennt die Aktivität von *Worker Node*s und verteilt fehlgeschlagene Aufgaben von im laufenden Betrieb ausgefallenen Maschinen an verfügbare Verarbeitungsknoten. Alle bis dahin noch nicht bestätigten Tupel werden ausgehend von dem produzierenden *Spout* wieder in die Verarbeitungskette eingespielt. Der auf den *Worker Node*s aktive *Supervisor daemon* überwacht die auf der jeweiligen Maschine aktiven Arbeitsprozesse und startet diese im Fehlerfall umgehend neu.

Die *Storm* Technologie kann für die Vorverarbeitung von Datenstrukturen verwendet werden und ist daher für die vorliegende Arbeit von Relevanz. Im Bezug auf die in Kapitel 5 gezeigte Architektur, kann dieses Framework zur Generierung von Eingangsdaten des Brokersystems angewandt werden.

Je nach Volumen und Heterogenität der auftretenden Datenquellen eines Anwendungsfalles ist zwischen der Direkten Verarbeitung durch *Storm* und einer Vorsortierung und Einordung in *Queue*s abzuwägen. Mit dem durch *Apache Kafka* bereitgestellten verteilten und robusten Nachrichtensystem sind hohe Datenvolumen zwischenzuspeichern. Dabei sind eingehende Datenquellen anhand der Werte von beherbergenden Attributen eines Datensatz in unterschiedliche *Topic*s einzuteilen, die von Konsumenten gelesen und eventuell in weitere Kategorien abgebildet werden. Das in Abbildung

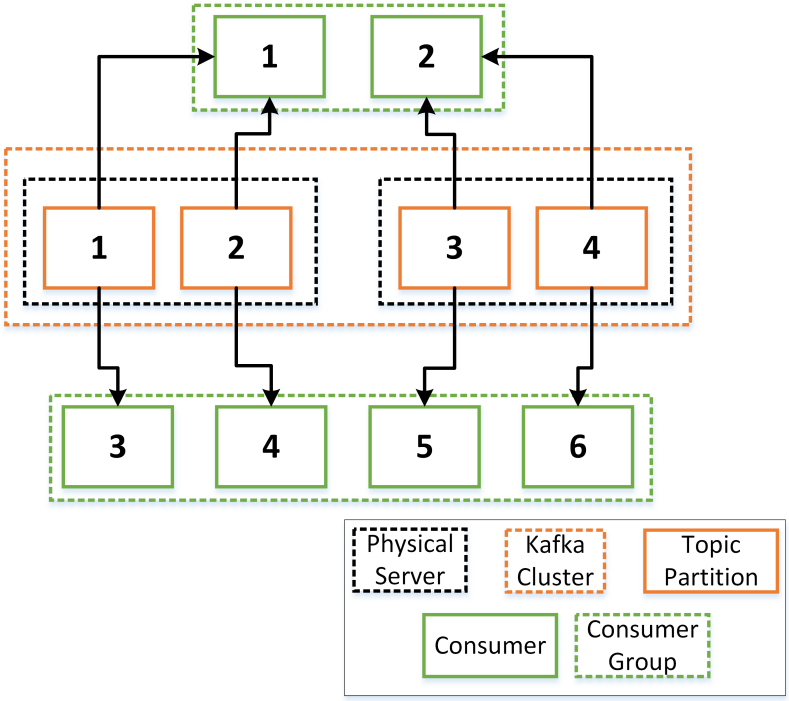

Abbildung 20.: Typisches Anwendungsbeispiel von Apache Kafka

20 aufgezeigte Anwendungsszenario verdeutlicht die Funktionsweise der Skalierbarkeit und Parallelität von *Kafka*. Ein *Kafka* Cluster besteht prinzipiell aus einer oder mehreren physischen Maschinen. Mit der Partitionierung einer Topic werden prinzipiell alle notwendigen Kommunikationsmuster auf unterschiedliche Ressourcen aufgeteilt. Die auf mehreren Maschinen platzierten Partitionen einer Topic beugen einer gegenseitigen Behinderung der Lesevorgänge von zusammengefassten Konsumenten einer *Consumer Group* vor. Der in einem verteilten System entstehende Netzwerkverkehr wird auf diese Weise auf alle dem Cluster zugehörigen Knoten gleichmäßig verteilt. Jede Partition wird ausgehend von einem lesenden und schreibenden *leader* über mehrere Maschinen als *follower*-Partition zur Wiederherstellung von Daten im Fehlerfall des *leader*s repliziert. Ein die *Topic* füllender *Producer* ist für die Platzierung von Nachrichten in einer der verfügbaren Partitionen verantwortlich und kann im Stile von *Round Robin* oder durch die Assoziation von Werten eines Nachrichtenattributs mit einer bestimmten Aufteilung agieren.

Mit Konsumenten, wie beispielsweise einer *KafkaSpout* Implementation in *Storm*,

kann eine Entkopplung der produzierenden Datenquelle von der verarbeitenden Topologie stattfinden. Eine Anwendung von Vorsortierung und Bearbeitung der Echtzeitdaten mit den beschriebenen Technologien steigert den Durchsatz eines auf die Verarbeitung bestimmter Ereignisse abgestimmten Systems durch Vermeidung von unnötiger Kommunikation. Die Kombination dieser beiden Technologien ist auf Grund von Skalierungsmechanismen eine performante Möglichkeit des Datenaustauschs zwischen den Endpunkten einer *Kafka Topic* und *Storm Topoligy*.

4.6. Verteilte Kommunikation heterogener Technologien

In vielen Fällen sind sprachenunabhängige Lösungen ohne den Einsatz zusätzlicher *Middleware* für die Umsetzung einer Unternehmung notwendig. Beispielsweise sind bei der Umsetzung von netzwerkbasierten Funktionsaufrufen mittels Remote Method Invocation (RMI) meist alle involvierten Aktoren (clients/servers) an die Implementierung in *Java* gebunden. Etablierte Technologien, wie Common Object Request Broker Architecture (CORBA) ermöglichen ebenfalls den entfernten Funktionsaufruf von Code. Die gewählte Hochsprache spielt auf Grund der Übersetzung eines *Request* in allgemeingültige Objekte und der Zuweisung an entsprechende Komponenten eine untergeordnete Rolle. Der durch die Vermittlung von Anfragen entstehende Flaschenhals des zentralen CORBA-*Broker*s ist ein Nachteil, der in dynamisch skalierten Infrastrukturen meist nicht tolerierbar ist.

Die Verknüpfung unterschiedlicher Module, welche in verschiedenen Programmiersprachen implementiert wurden, kann mittels Rahmenwerken wie *Apache Thrift* in dezentralisierter Form ermöglicht werden[ASK07]. Der Bezug von *Apache Thrift* zu dieser Arbeit besteht nicht nur in der möglichen Integration der Technologie in einen *Broker*, sondern birgt parallelen zu den in 4.1 beschriebenen Vorgehensweise. Mit der *Thrift* Interface Definition Language (IDL) wird eine gemeinsame Referenz von Schnittstellen des Dienstes durch Annotation von Datenstrukturen festgelegt. Darin verwendbaren Basistypen, wie *bool, byte* oder *i32*, repräsentieren die gemeinsame Grundlage unterschiedlicher Programmiersprachen. Unter Verwendung dieser Datentypen lassen sich *struct*s definieren, die eine gemeinsame Objektstruktur durch ihre Felder bestimmen und Sprachenübergreifend verwendet werden können. Neben primitiven Datentypen existieren ebenfalls *Container*s, wie geordnete Listen, ungeordnete Sets und Maps. Eine ausformulierte IDL Datei kann, unter Verwendung eines *Code Generator*s in Form von `thrift-gen [programming language] [idl file]`, in die entsprechende Sprache transformiert werden. Dabei ist der erste Parameter mit der Auswahl eines *Template*s und die *.idl* Datei mit der Instanz einer DSL zu vergleichen.

Der *network stack* von *Thrift* beruht auf drei aufeinander aufbauenden Schichten. Innerhalb der Transportschicht sind Details der physischen Kommunikation einzelner

Module festgelegt. Der unterliegende Kommunikationsmechanismus greift entweder auf Transmission Control Protocol (TCP)/IP Stream Sockets, *in-memory* Rohdaten oder Dateien zurück und ist die gemeinsame Schnittstelle für den bidirektionalen Austausch von Informationen involvierter Module. Der Entwickler einer Anwendung kann deren Logik dabei unabhängig von der gewählten Interaktion implementieren. Das *Transport Interface* ist ebenfalls durch objektorientierte Techniken wie Kompositionen erweiterbar. Die Protokollschicht separiert die für einen Informationsaustausch verwendeten Datenstrukturen von der Transportschicht-Repräsentation. Für den Effizienten Zugriff auf große Datenstrukturen ermöglicht die Protokollschnittstelle einen sequenzierten Nachrichtenaustausch, sowie die Kodierung von Basistypen, Containern und Strukturen in Funktionen. Das *Processing Layer* ist für das herunterbrechen von verteilten Mechanismen in Ein- und Ausgangsdatenströme verantwortlich. Im Bezug auf die Interaktion von Client und Server beseitigt diese Schicht jegliche Hürden während der Implementierung. Die generierten Module und Schnittstellen bieten dabei unter anderem die Möglichkeit der Anwendung von asynchronen Kommunikationsmustern durch clientseitige Definition von *callback*s und der serverseitigen Anwendung von *Pools* oder *Threads*. Der Einsatz solcher Technologien ist auf Grund der steigenden Komplexität einer Unternehmung in jedem Fall gegen den möglichen Mehrwert abzuwägen.

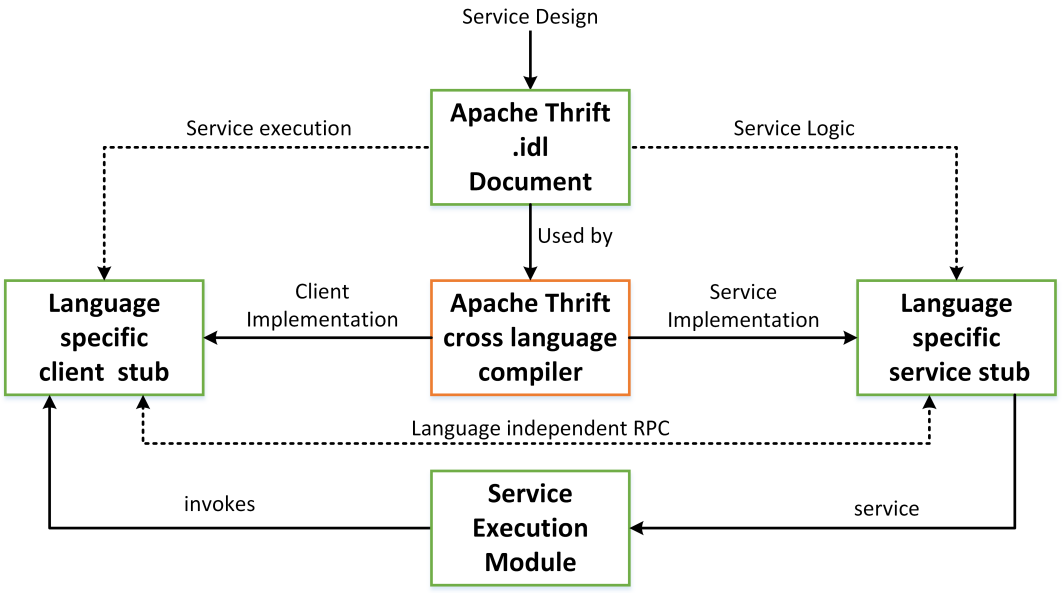

Abbildung 21.: Typischer Apache Thrift Workflow
Eigene Darstellung in Anlehnung an [ASK07, Rak15]

Abbildung 21 stellt den typischen Anwendungsfall einer auf *Apache Thrift* basierenden Anwendung dar. Unter Verwendung von in der *Thrift* IDL definierten Typen können beliebige Dienste deklariert werden. Die Parametrisierung von einzelner Remote Procedure Call (RPC)s resultiert durch den *Compiler* generierten Schnittstellen,

welche innerhalb des jeweiligen Moduls in einer spezifischen Programmiersprache auszuformulieren sind. Je nach Anwendungsfall kann die serverseitige Implementierung in blockierender oder parallel agierender Weise betrieben werden. Das gewählte Interaktionsmuster ist in vielen Fällen ausschlaggebend für die Kompatibilität von definierten Diensten mit weiteren Technologien oder problemspezifischen Anforderungen und dementsprechend zu wählen.

Besteht ein Projekt aus vielen in unterschiedlichen Hochsprachen formulierten Modulen, sind generierte Client-Strukturen in das jeweilige *Service Execution Module* an entsprechender Stelle einzubetten. Der entfernte Funktionsaufruf durch diesen Client kann mit Hilfe einer in *Thrift* spezifizierten Serialisierung von Attributen ohne weiteres Zutun und unter Verwendung von nativen Typen der gewählten Programmiersprache parametrisiert werden. Aus Sicht der anwendenden Komponente ist der ausgeführte Dienst hinter einem lokalen Funktionsaufruf abstrahiert.

Im Bezug auf Python, kann neben den offiziellen Paketen auch die *thriftpy* Implementierung[4] verwendet werden. Mit der Anwendung dieser Bibliothek ist die Übersetzung von IDLs durch den *Thrift Compiler* in entsprechende Implementierungen nicht weiter notwendig. Die Programmlogik verweist dabei mit einem `thriftpy.load` Funktionsaufruf auf die jeweilige IDL. Mit der in *thriftpy.rpc* bereitgestellten *make_server* Funktion und der entsprechenden Implementierung der IDL Diense in einer *Dispatcher* Klasse, ist die serverseitige Definition des Dienstes in wenigen Zeilen umsetzbar. Die *make_client* Funktion resultiert in einem entsprechenden Client, welcher die entfernte Implementierung, anhand des angegebenen Sockets, als Funktionsaufruf ermöglicht. Die Anwendung dieser Bibliothek verringert die Komplexität der Implementierung einer Thrift-basierten Anwendung erheblich, da der in Abbildung 21 gezeigte Workflow nicht von der *Cross Language Compiler* Komponente abhängig ist.

[4] `https://github.com/eleme/thriftpy`

5. Generische Implementierung der Broker Architektur

Das vorliegende Kapitel diskutiert Details einer generischen Architektur für problemspezifische, auf dynamische Eingangsdaten abgestimmte, Plattform*Broker*. Da sich Softwarearchitekturen aus Strukturen des Softwaresystems zusammensetzen, welche wesentliche Elemente und deren Interaktionsbeziehungen beschreiben, eignet sich diese übergeordnete Abstraktionsebene besonders für die Analyse von damit realisierbaren Implementationen [Lan04]. In den folgenden Abschnitten wird häufig der Bezug zu modellgetriebener Softwareentwicklung hergestellt. Dabei liegt das Hauptaugenmerk auf der Allgemeingültigkeit von Konzepten und der Abbildung von konfigurierbaren Prozessen innerhalb verarbeitenden Plattformen. Das Alleinstellungsmerkmal dieser *Broker* Architektur besteht aus der Kombination von definierten PaaS Mechanismen zur Vermittlung von Datensätzen an die problemspezifische Aufgabenstellung ausführende virtualisierte Softwaresammlungen. Der Betrieb einer solchen Vermittlungsschicht rechtfertigt sich durch komplexe Strukturen von Daten und deren Verarbeitung innerhalb isolierter Laufzeitumgebungen.

Neben grundlegenden Anforderungen an Geltungsbereiche der in die Vermittlungsschicht involvierten Komponenten, werden auf die Datenquelle angewandte Muster zur Aufgabenumsetzung in Abschnitt 5.1 beschrieben. Eine in Unterkapitel 5.2 aufgezeigte Modularisierung der *Broker* Architektur ermöglicht die Umsetzung der funktionalen Eigenschaften unter Verwendung dynamischer Technologien. Darauf folgend wird auf die Platzierung der vorab aufgezeigten Strukturen und deren Betrieb in verteilten Systemen unter Verwendung generischer Laufzeitumgebungen eingegangen. Die Implementierungen dieser Strukturen ist nachfolgend in Abschnitt 5.4 an beispielhaft ausgewählten Technologien veranschaulicht. Eine Grammatik zur effizienten Erstellung dieser vermittelnden Schichten in Abhängigkeiten von Eingangsdaten wird in Abschnitt 5.5 vorgestellt. Die Aufgabenumsetzung einer generischen Architektur geschieht unter Verwendung konkreter *Cloud Image*s, deren Funktionsweise und Konfiguration mit dynamischen Eingangsdaten diskutiert wird. Abschließend werden Aufgabenbereiche der in einen domänenspezifischen *Broker* involvierte Aktoren in Abschnitt 5.6 identifiziert. Für die konkrete Anwendung dieser *Broker* Architektur wird auf Kapitel 6 verwiesen.

Während der Allokation einer VM kann diese, unter Verwendung eines für den Anwendungsfall vorab generierten *Cloud Image*, mit konkreten Eigenschaften granular angepasst werden. Das Betriebssystem eines virtualisierten Servers ist dafür mit ei-

ner an die Problemstellung angepassten Software zu bestücken, ebenfalls muss dieses *Image* mit einer Möglichkeit zur Entgegennahme von Lokalitäten der Datenquelle ausgestattet sein. Beispielsweise können Parameter wie IP Adresse oder DNS-*hostname* einer zu überwachenden Ressource innerhalb einer *Monitoring* Software deklariert werden.

Da eine VM in einem solchen Szenario mit wenigen Aufgaben betraut ist, wird zwar die der Virtualisierung der zugrundeliegenden Hardware bei steigenden Anfragen unter zusätzliche Last gesetzt, ausführende Instanzen agieren jedoch mit minimalem Overhead auf Systemebene. Innerhalb des CMS kann ungewöhnlichen Aktionsmustern durch die Vergabe von Rechten und der Definition von Regeln über vorhandene Ressourcen gegengesteuert werden. Zu instanziierende VMs können mit unterschiedlichen Anwendungen und Architekturen ausgestattet und auf den jeweiligen *Use Case* adaptiert werden.Die Sinnhaftigkeit der Allokation virtueller Ressourcen für die Bereitstellung von isolierten Verarbeitungsplattformen ist dabei immer von der Problemstellung abhängig. Im Folgenden wird jedoch von exklusiven Verarbeitungsplattformen ausgegangen.

5.1. Virtualisierung von Verarbeitungsplattformen mit Broker Strukturen

Die Verarbeitung von Daten kann heutzutage durch existierende Werkzeuge meist in automatisierter Art und Weise durchgeführt werden. Abhängig von der definierten Aufgabe und dem resultierenden Ressourcenverbrauch, verringert sich die Performanz der handelnden Maschine mit steigender Anzahl von parallel zu bearbeitenden Datensätzen. Das bereitgestellten von Plattformen in einer durch Virtualisierung von einander isolierten Infrastruktur ermöglicht eine dynamische und skalierbare Grundlage der problemspezifischen Verarbeitung von Daten. Korrekte Konfigurationen und Einbettungen dieser *Jobs* in bereits vorhandene Infrastrukturen bzw. Hypervisor sind eine Hürde, die für den Endbenutzer transparent gehalten werden muss, da hierbei im Extremfall von technischen Laien auszugehen ist. Die Kombination traditioneller Cloud *Broker* und einer anwendungsspezifischen Konfiguration des angebotenen Dienstes mit einer Datenquelle ist in Abbildung 22 aufgezeigt. Durch die gegebenenfalls automatisierte Konfiguration und bereitgestellten Schnittstellen zur Ausführung einzelner Anwendungen vermischen sich dabei die Grundzüge von PaaS Diensten mit denen eines IaaS.

Aus Sicht eines *Data Analyst* kann eine Verarbeitungsplattform für spezifische Datenquellen aus unterschiedlichen, mit einander interagierenden, Anwendungen bestehen. Diese in einem *Platform Image* zusammengefassten Werkzeuge, beziehen sich auf einen konkreten Anwendungsfall und können daher mit Eigenentwicklungen angereichert oder in einem Workflow aggregiert werden. Ebenfalls ist die Bereitstellung

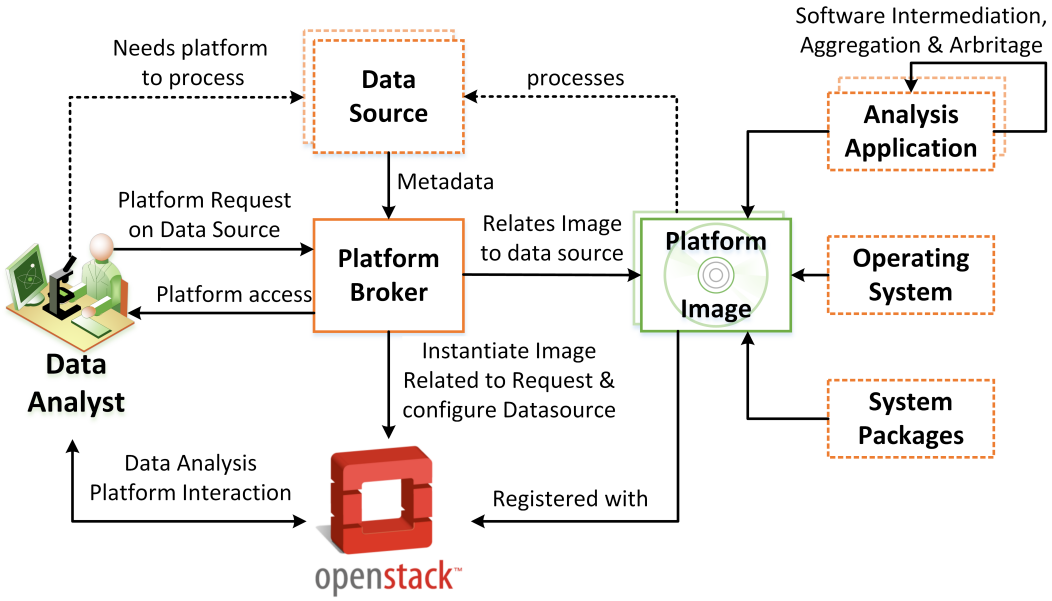

Abbildung 22.: Verarbeitung von Rohdaten durch virtualisierte Verarbeitungsplattformen

unterschiedlicher Anwendungen, die auf einzelne Arbeitsschritte abgestimmt sind und möglicherweise in bestimmter Reihenfolge ausgeführt werden müssen, denkbar. In jedem Fall steht dem Endanwender die Wahl der entsprechenden Software, deren Einbettung in den konfigurierbaren Gesamtablauf des Analysevorgangs gewährleistet sein muss, während der Definition eines *Broker*s frei. Abhängig von dem gewählten Ökosystem an Anwendungen ist ein Betriebssystem zu wählen und mit grundlegenden Softwarepaketen zur Aufgabenbewältigung, wie grafischen Oberflächen oder bestimmten Kompilierungen verschiedener Technologien, anzureichern. Die Formulierung des *Plattform Image* kann bei genauerer Betrachtung auch alleinstehend mit den in Abschnitt 2.3 aufgezeigten Paradigmen einer Domäne verglichen werden. Das Betriebssystem ist dabei die allumfassende und nicht änderbare Plattform, in dem Regelwerke der Konfiguration von einzelnen Anwendungen in unterschiedlichen Varianten vordefiniert sind. Die Auswahl und Verknüpfung der verarbeitenden Programme ist verwandt mit dem individuell zu deklarierenden Anteil der domänenspezifischen Endanwendung.

Der *Broker* einer solchen Analyseplattform ist in der Lage unterschiedliche Datenquellen auf das passende Cloud Image abzubilden. Dieses muss für die spätere Instanziierung mit einem CMS registriert und für den *Broker* erreichbar sein. Die Übersetzung von Metadaten einer bestimmten Domäne in den initialen Konfigurationsprozess einer Plattform, ist in jedem Fall problemspezifisch und wird im folgenden mit ausgewählten MDSD Paradigmen argumentiert. Unveränderliche Eigenschaften der Domäne eines Plattform *Broker*s sind die originalen Datenquellen und das verwendete CMS. Die individuellen Teile der Problemdefinition beziehen sich auf gewählte Verarbeitungsschritte unter Verwendung spezifischer Software. Um vordefinierte Aktionen und Regeln ei-

ner Vermittlungsschicht zu gewährleisten sind alle Variationspunkte, im Bezug auf die vorgegebenen CMS- und Plattforminteraktionen, sowie Anfragen an eine Datenquelle, in Form von ausformulierten und auf die *Execution Engine* abgestimmten Strukturen bereitgestellt. Für die angebotene Benutzerschnittstelle werden ausgewählte Teile dieser auf einander aufbauenden Schichten, innerhalb entfernt ausführbaren und parametrisierbaren Strukturen, kombiniert.

Die Platzierung der involvierten Komponenten und Aktoren ist nicht explizit dargestellt, da ein *Platform Broker* durch die modulare Verwendung von Verbindungsmodulen mit unterschiedlichen Technologien interagieren kann. Die physische Verteilung der Komponenten eines *Platform Broker* Szenario ist ebenso, wie verwendete *Platform Images* von der problemspezifischen Datenquelle abhängig, nicht allgemeingültig und während der Implementierung der Anwendungslogik durch den Entwickler zu beachten. Die eigentliche Dienstleistung des *Platform Broker* muss daher an die vorherrschenden infrastrukturellen Begebenheiten angepasst sein und bezieht sich auf die im folgenden erläuterten Fälle:

1. Alle Aktoren sind in unterschiedlichen Cloud Umgebungen angesiedelt. Der *Platform Broker* kann als eigenständiger Dienst betrachtet werden, welcher den IaaS eines externen CSP für die Bereitstellung von Verarbeitungsplattformen verwendet. Das durch den *Platform Broker* erstellte *Cloud Image* ist auf die in einer separaten Cloudumgebung platzierten Datenquelle angepasst und mit dem IaaS Anbieter registriert. Dabei kann der *Platform Broker* als klassischer Vermittler von verteilten Diensten unter Betrachtung von eigenen Referenzen auf dynamische Eingabedaten aus externen Quellen verglichen werden.

2. Ein IaaS Anbieter verwendet den *Platform Broker* als zusätzlichen Dienst innerhalb seiner eigenen Infrastruktur. In dieser Variante kann die Vermittlungsschicht die Definition und Umsetzung einer Problemstellung durch die Verwendung von *Hybrid Cloud* Konzepten von einander trennen. Auf Grund der Konzentration von Ansätzen zur Problemlösung durch einen exklusiven CSP ist die Einbettung dynamisch erstellter Plattform Instanzen in eine bereits bestehende, nicht öffentlich zugängliche Verarbeitungsinfrastruktur möglich. Unter Einsatz von den in Abschnitt 4.5 beschriebenen Big-Data Systemen sind gewonnene Informationen aus bisher unternehmensinternen Fachbereichen, für die Anreicherung eines Dienstes mit einem Mehrwert bei dessen Aufgabenumsetzung, verwendbar. Die eingehende Datenquelle kann beispielsweise aus mehreren externen Informationsanbietern bestehen und mit Attributen bzw. den Relationen der jeweiligen Quellen innerhalb einer Domäne angereichert werden. Die Bestimmung von Abhängigkeiten der Eingangsdaten kann sich ebenso auf ein fest definiertes Geschäftsmodell beziehen, in dem keine Vorverarbeitung von

Informationen benötigt wird. Die Verwendung und Platzierung von Komponenten des *Broker* Dienstes bleibt, wie auch die Definition der Plattform, für den Anwender transparent. Durch direkten Zugang zur Virtualisierungstechnologie eines CSP sind zusätzliche Möglichkeiten im Bezug auf Kommunikationsmechanismen des *Broker*s, bzw. des *Platform Image*s, mit Drittsystemen gegeben. Dieser Vorteil spiegelt sich auch während der Erstellung eines *Platform Image* wieder, bei der bereits eine erste Anpassung an die grundlegende Infrastruktur der *Private Cloud* des CSP vorgenommen werden kann.

3. Der Anbieter einer Datenquelle ermöglicht seinen Kunden einen direkte Verarbeitung durch die Anwendung eines *Platform Broker*. Da die Entwickler der Vermittlungsschicht totalen Einblick in die Strukturen und tiefes Verständnis der angebotenen Daten besitzen, ist eine feingranulare Abstimmung der bereitzustellenden Verarbeitungsplattformen auf die Bedürfnisse der Kunden möglich. Die verarbeitende Infrastruktur ist dabei an einen externen CSP ausgelagert.

4. Der *Platform Broker* ist ein unternehmensinterner Mechanismus für die Bereitstellung und Umsetzung von Workflows in einer virtualisierten Infrastruktur. Da alle Komponenten innerhalb einer *Private Cloud* agieren ist der gesamte Ablauf, von der Transformation problemspezifischer Metadaten in Elemente der Benutzerschnittstelle über die Definition des *Platform Image* bis hin zur Instanziierung und Konfiguration von Plattformen, auf interne Geschäftsprozesse abbildbar. Hierbei ist die Platzierung von Plattformen in der physischen Nähe der zu verarbeitenden Datenquelle zu diskutieren. Die dadurch verringerte Netzwerklast kann bei der Übertragung massiver Datenmengen zu einer effizienten Parallelisierung unterschiedlicher Plattformen beitragen.

Anforderungen an die Architektur Mit der im folgenden Vorgestellten Architektur soll eine Erkennung und Vermittlung von Datenstrukturen und deren Zuweisung an verarbeitende und virtualisierte Plattformen umgesetzt werden. Eine Verwendung beliebiger Virtualisierungstechnologien soll dem Anwendungsentwickler ein flexibles Modell zur Definition von Vermittlungsschichten bereitstellen. Dabei ist die dynamische Erweiterung und Konfiguration von Broker Instanzen um zusätzliche oder sich ändernde Plattformen oder Datenquellen gewährleistet. Die Erweiterbarkeit und Wiederverwendbarkeit einer Architektur ist in gewissem Maße an die Allgemeingültigkeit einer spezifizierten Problemstellung oder Domäne gekoppelt. Dennoch sollen die folgenden Eigenschaften zur Flexibilität bei der Formulierung der spezifischen Broker eingehalten werden.

1. Eine modulare Erweiterbarkeit und Austauschbarkeit von Komponenten der Architektur um beliebige Technologien soll jederzeit umsetzbar sein. Der program-

matische Eingriff in grundlegende Strukturen der Brokerlogik ist dabei zu vermeiden.

2. Die Wahl der Laufzeitumgebungen des *Broker*s soll durch abstrahierte Strukturen der Architektur definierbar sein. Eine Definition von *Templates* und deren Instanziierung durch Code Generatoren soll dem Entwickler die dynamische Wahl einer grundlegenden Technologie der Vermittlungsschicht ermöglichen. Mit der Erweiterbarkeit der *Templates* soll die Anwendung des Brokers im Hinblick auf die Integration in bereits bestehende Anwendungen ermöglicht werden.

3. Die Integration beliebiger Anwendungslogik soll bei jeder möglichen Kommunikation von Modulen innerhalb des Brokers ermöglicht werden.

Bedeutung von Datenquellen für die Aufgabenumsetzung

Die grundlegende Definition von Metadaten einer Datenquelle ermöglicht dem Endanwender des *Broker*s die Delegation von verschiedenen Datenverarbeitungsaufgaben an virtualisierte Plattformen. Informationen über zu bearbeitende Strukturen sind neben tiefem Verständnis über die Aussagekraft von Daten und einem konkretem Plan zur Umsetzung der Aufgabenbewältigung spezifizierter Fragestellungen Ausgangspunkt der problemspezifischen Vermittlungsschicht. Die Umsetzung einer konfigurierbaren Plattform ist auf die Kommunikation mit vorkommenden Technologien der Datenhaltung angewiesen und dementsprechend zu präparieren.

Wie in Abbildung 23 dargestellt, kann die dezentrale Verarbeitung von Datenstrukturen in drei Schichten abstrahiert werden. Die eigentlichen Daten werden durch *Metadaten* beschrieben und in einem weiteren Schritt auf für den Endbenutzer relevante Eigenschaften heruntergebrochen. Die Aussagekraft komplexer Strukturen von zu analysierenden Rohdaten ist nicht immer auf den ersten Blick ersichtlich, weswegen an dieser Stelle unterschiedliche Technologien zur Vorverarbeitung und Kategorisierung der Datensätze in den Prozess mit eingebunden werden können. Ein auf diese Struktur angepasster *Broker* ist durch eine im Hintergrund gepflegte Datenstruktur in der Lage, die ursprünglichen Rohdaten innerhalb einer Aktion aufzulösen. Dabei implementierte Aktionen sind als Verbindung des Brokers mit Virtualisierungstechniken zu betrachten, innerhalb derer die eigentliche Verarbeitung *externer* Rohdaten geschieht. Die Instanziierung einer Verarbeitungsplattform beinhaltet deren Vorkonfiguration mit Metadaten der jeweiligen Ursprungsdaten, wie beispielsweise deren URL, welche den ausführenden Anwendungen als Startparameter übergeben werden kann. Die eingehenden Rohdaten stehen in Beziehung zu problemspezifischen Aufgabenstellungen und sind daher durch darauf angepasste Werkzeuge zu verarbeiten. Eine Rückkopplung der virtualisierten Bearbeitungswerkzeuge verschafft dem Benutzer des Brokers zusammenfassende Informationen über die momentan existierenden Anwendungen so-

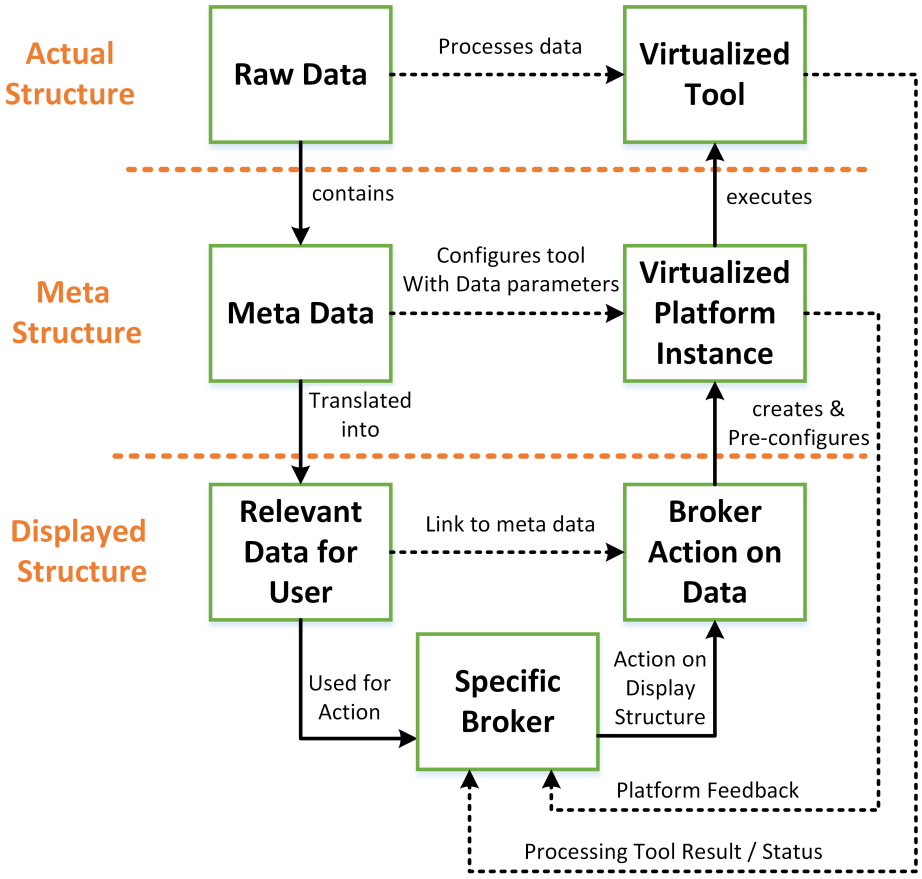

Abbildung 23.: Verarbeitung von Rohdaten durch virtualisierte Anwendungen

wie deren Zwischenergebnisse. Ebenfalls können ressourcenspezifische Informationen der ausführenden VM dargestellt werden.

Verarbeitung von Daten

Die Verarbeitung von durch einen Broker interpretierten und an eine exklusive Verarbeitungsplattform vermittelten Daten, stellt die Grundproblematik der vorliegenden Arbeit dar. Die Verbindung zu einer Datenquelle muss demnach nicht nur in der verarbeitenden Ressource, sondern auch in der Vermittlungsschicht hergestellt werden. Handelt es sich dabei um bereits durch vorhergehende und möglicherweise von der Problemstellung des Brokers unabhängig aufbereitete Datensätze, ist diese Komponente mit Anfragen an die verwendeten Technologien auszustatten. Ohne ein solches Aufbereitungssystem kann die Aussagekraft der in den Metadaten existierenden Attribute nicht gewährleistet werden. Eine Benutzeroberfläche zeigt dabei alle der Metadatenstruktur entsprechenden Datensätze einer Quelle an. Mit der tabellarischen Auflistung von existierenden Datensätzen wird der Einstiegspunkte zu deren Weiterverarbeitung in individuell formulierbaren Aktionen repräsentiert. Diese sind nicht zwangsläufig mit CMS-Funktionalitäten zur Erstellung einer virtuellen Ressource zu assoziieren. Je nach Bedeutung eines Attributs, sind REST calls oder RPC Aufrufe zu

weiteren Systemen denkbar. Die Darstellung einzelner Zeilen ist dabei durch die Anwendungslogik anhand der Werte von darin enthaltenen Attributen weiter einschränkbar. Ebenfalls können Datensätze auf dieser Ebene, in Abhängigkeit von beispielsweise einem Zeitfenster, als einzelne Zeile zusammengefasst werden. Diese Verknüpfung von Informationen unterschiedlichen Ursprungs kann in einer verarbeitenden Plattform zu einer höheren Aussagekraft der Problemstellung beitragen.

Das Festlegen solcher Vermittlungsmuster ist immer abhängig von der Beständigkeit der Strukturen einer Datenquelle. Ändern sich diese während eines Projekts, ist neben der Darstellung in der Benutzerschnittstelle auch die Automatisierungen der Erstellung einer Verarbeitungsressource bzw. deren zu konfigurierenden Daten, darauf anzupassen. Mit den in Abschnitt 4.5 beschriebenen Technologien können die Eingangsdaten im Stil einer Vorverarbeitung zusammengefasst, aggregiert und für die weitere Verarbeitung zugänglich gemacht werden. Dabei ist eine Änderung des gewählten Datenmodells eher unwahrscheinlich. Durch den Broker erstellte Plattformen sind in einer Datenstruktur zu pflegen, die in einer weiteren Sichtweise für Aktionen über diese verwendet wird.

5.2. Modularer Architekturaufbau des Plattform Brokers

Mit der im folgenden vorgestellten Architektur eines *Platform Broker*s sind vielfältige Anwendungsszenarien umsetzbar. Die Konzepte und Modelle einzelner Komponenten des *Broker*s sind prinzipiell generisch implementierbar und legen den Grundstein für die Erweiterbarkeit von Instanzen einer Vermittlungsschicht. Ein modularer Aufbau der Implementation gewährleistet neben der Einbettung von individuell ausformulierter Funktionalität auch die Verwendung beliebiger Virtualisierungstechnologien und Datenquellen.

Die in Abbildung 24 dargestellten Module unterteilen die Architektur in voneinander unabhängige Implementationen. Bei der Inbetriebnahme des *Broker*s kommunizieren von einander abhängige Komponenten über technologiespezifische Mechanismen, deren Anwendungslogik an entsprechender Stelle auszuformulieren ist.

Ein *Broker Task* bezieht sich auf Metadaten beliebig platzierter Datenquellen und wird innerhalb des *Task Definition Module* ausformuliert. Der grundlegende Verbindungsaufbau zu Technologien der Datenquelle ist in einem *Connector* Modul festzuhalten und kann beliebig komplexe Züge annehmen. Die Aggregation und Filterung mehrerer verteilter Datenquellen stellt den Extremfall dar und konzentriert die Eingangsdaten innerhalb eines Moduls. Eine Darstellung der Metadaten innerhalb des *User Interface* ist auf aussagekräftige Attribute zu beschränken und in eigenen Repräsentationen abzubilden. Diese stellen den eingehenden Zustand der Vermittlungsschicht aus Sicht des Endanwenders dar und werden bei der Verwaltung des Metada-

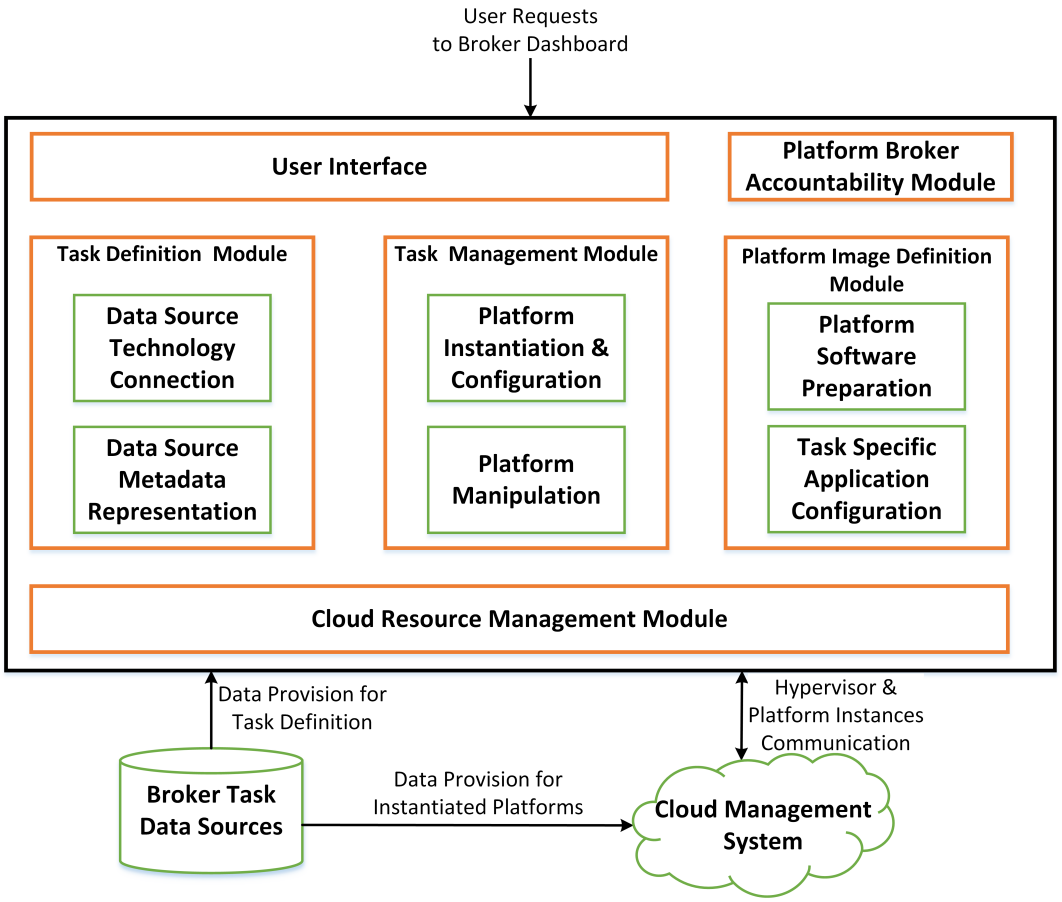

Abbildung 24.: Module des Cloud Ressourcen Brokers

tenindex der Datenquelle in programmatischer Objektform für die Darstellung im *User Interface* verwendet. Im einfachsten Fall wird der Pfad zu einer geläufigen Datenstruktur wie JSON oder XML deklariert, in denen ausgewählte Attribute eine Bedeutung für die Definition der individuellen Problemstellung besitzen. Beispielsweise stellen empfangene Informationen den Index eines Web-Archivs dar, in dem keine Änderung an bereits bestehenden Dokumenten vorgenommen wird. Die als relevante Informationen identifizierten Teile können beschreibende Felder des jeweiligen Dokuments oder Verweise auf den physischen Ort der Datei sein. Diese Attribute können innerhalb einer Relation der *Broker* Datenbank abgebildet werden, auf die in einem solchen simplen Szenario zurückzugreifen ist. Besteht eine Anforderung an die Verfolgung von sich ändernden Attributen der Datensätze, wird in der letztendlichen Implementierung jedoch auf Strukturen zum Verbindungsaufbau mit externen Systemen zurückgegriffen. Der Einsatz von unternehmensinternen eigenständigen Systemen ist für das *preprocessing* der Daten im Stil von *Data Warehouse*s bzw. Spark oder Storm Topologien unter Verwendung beliebiger Technologien der Datenhaltung denkbar. Bei der Darstellung von Echtzeitdaten ist diese zusätzliche Komponente mit zyklischen Abfragen an die Quelle und dem festhalten von Änderungen im Zielspeicher der jeweiligen Datenhaltung als vom *Broker* unabhängige und nebenläufige Anwendung auszuführen. Innerhalb

des *User Interface* wird daher in diesem Fall auf eine Repräsentation der momentan vorhandenen oder historischen Metadaten eines externen Systems verwiesen.

Das *Task Management Module* beinhaltet die Logik zur Transformation von Benutzerereignissen auf Metadaten in Aktionen der durch den *Broker* gepflegten Infrastruktur. Die eigentliche Aufgabenumsetzung eines *Broker Task* besteht aus der Instanziierung von Plattformen und deren Konfiguration mit einer spezifischen Datenquelle anhand vorab ausgewählter Metadaten. Diese zentrale Funktionalität ist die Grundlage für den ausgehenden Zustand des *Broker*s und wird daher als *Broker*-interner Plattform-Metadatenindex persistiert. Die automatische Instanziierung bzw. Anwendung von beliebiger Funktionalität ist ebenfalls in Abhängigkeit von eingehender Metadaten des *Task Definition Module* definierbar. Beispielsweise ist das überschreiten bestimmter Schwellwerte von Attributen einer Datenquelle ein Indikator für Anomalien in den Rohdaten und kann mit der automatischen Erstellung einer Analyseplattform, sowie der Benachrichtigung dieses Vorgangs an einen Analysten beantwortet werden. Derartige Umsetzungen können unter Umständen das zyklische Abfragen der Datenquelle verlangen. Abhängig von der zugrundeliegenden Technologie sind asynchrone *callback* Funktionen in separaten Modulen denkbar. Grundlegend realisiert der *Broker* jedoch die Erstellung von Plattformen auf Anfrage der Endbenutzer. Die Manipulation vorhandener Instanzen basiert prinzipiell auf der Kommunikation mit Hypervisor Technologien, abhängig von der verwendeten Softwaresammlung einer Plattform ist auch die direkte Kommunikation mit der jeweiligen Anwendung als Aktion definierbar.

Das *Platform Image Definition Module* deklariert in erster Linie die Problemspezifischen und mit den Eingangsdaten verknüpften Anwendungen. Um die Aufgabenumsetzung bei der Erstellung einer Plattform zu beginnen, sind alle involvierten Programme auf einander abzustimmen. Je nach Komplexität der Datenstruktur kann der Fokus einer Plattform auf unterschiedlichen Aspekten liegen. Daher ist neben der Verwendung eines *One-Size-Fits-All* Ansatz der Plattform auch die Definition mehrerer, auf die jeweilige Problemstellungen angepasster, *Cloud Images* möglich. Abhängig von den gewählten Eingangsdaten kann die passende Plattform durch die Analyse bestimmter Attribute der Metadaten auf das entsprechende *Cloud Image* abgebildet und mit individuellen Parametern konfiguriert werden. Die innerhalb der Plattform platzierte Softwaresammlung verwendet die Übergabeattribute des *Brokers* für deren initiale Konfiguration und anschließende Aufgabenbearbeitung. Daher wird ein Mechanismus für die dynamische Konfiguration von *Task* spezifischen Anwendungen während der Instanziierung einer Verarbeitungsplattform bereitgestellt. Dieser füllt beispielsweise unterschiedliche Konfigurationsdateien oder startet Anwendungen mit bestimmten Parametern der ausgewählten Datenquelle. Das ausformulierte und kompilierte *Cloud Image* muss dem Hypervisor grundsätzlich bereitgestellt werden.

Da jegliche Hypervisortechnologie für die Instanziierung einer Plattform eingesetzt werden kann, sind grundlegende Aktionen innerhalb des *Cloud Resource Management Module* zu definieren. Die minimale Funktionalität umfasst hierbei das erstellen und entfernen von Instanzen eines Plattform *Image*.

Modellierung von Broker Tasks

Die Umsetzung von Problemstellungen dynamischer Daten ist individuell zu bestimmen und in den meisten Fällen nicht wiederverwendbar.

Eine durch den *Broker* zu bearbeitende Aufgabe besteht vielmehr aus automatisierter CMS Funktionalitäten unter dem Gesichtspunkt der Bereitstellung definierbarer Verarbeitungsplattformen für eine spezifische Datenstruktur. Die allgemeine Struktur eines *Broker Task* ist in Abbildung 25 dargestellt. Grundlegend besteht dieser aus zu instanziierenden Task Plattformen und darauf angepassten Benutzerschnittstellen, sowie einer Komponente zum Management von Benutzern und ausgeführter Funktionalität.

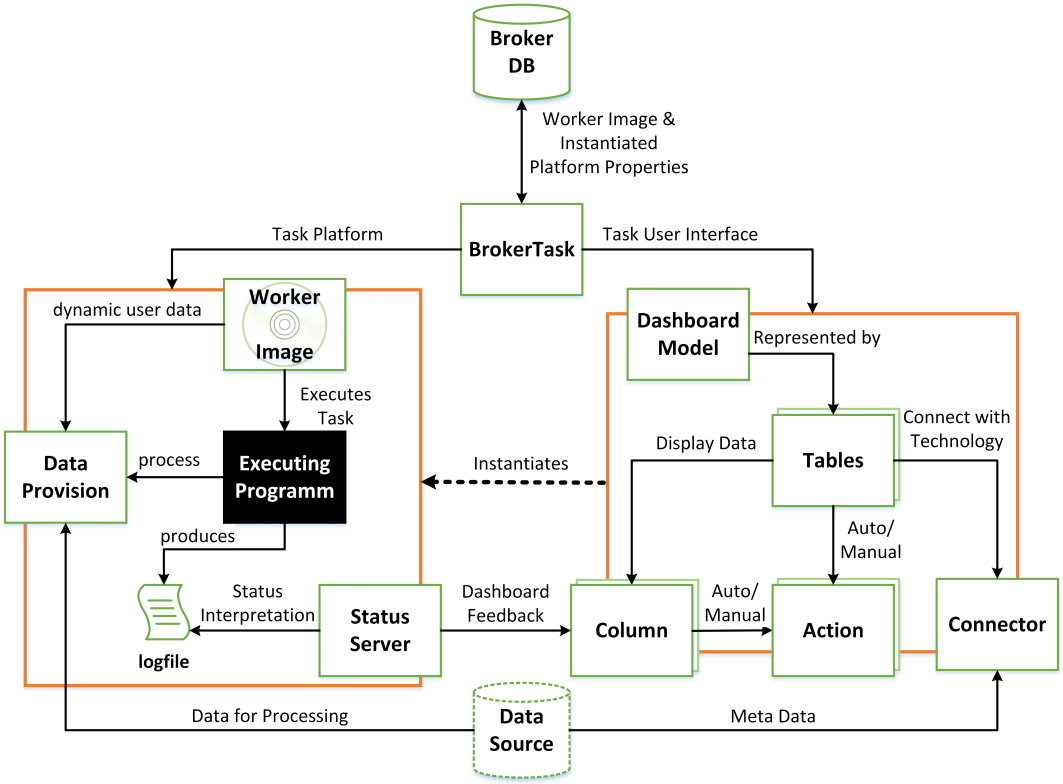

Abbildung 25.: Zusammenhänge von Steuerung und Verarbeitung von Daten durch Cloud Resourcen Broker

Das *Worker Image* beinhalt ein Modul zum dynamischen Verbindungsaufbau der Instanz mit einer Datenquelle. Dieser Mechanismus tritt während der Instanziierung einer Plattform in Kraft und wird in der Regel aktiv durch den Anwender, mit Referenzen auf spezifische Eingangsdaten der Vermittlungsschicht, bestimmt. Dabei wird

die grundlegende Annahme getroffen, dass einzelne Attribute der Metadaten eines Datensatzes ausschlaggebend für die Bereitstellung der eigentlichen Rohdaten sind. Wird beispielsweise auf eine physische Datei eines Network File System (NFS) verwiesen, kann diese automatisch beim Starten der VM durch Systembefehle *gemountet* werden. Möglicherweise ist die Platzierung der Rohdaten aber nicht direkt ersichtlich, sondern auf Umwegen über Programmlogik zu restaurieren. Eine Verknüpfung von Datensätzen mit der auszuführenden Plattform kann ebenfalls durch den parametrisierten Aufruf von eigenentwickelten Modulen oder der auszuführenden Anwendung selbst, realisiert werden. Ebenfalls kann sich eine Software auf Attribute einer Konfigurationsdatei stützen und selbstständig auf entfernte Daten zugreifen.

Der Betrieb einer Plattform kann auf mehreren, auf einander abgestimmten Anwendungen basieren. Sind mehrere Prozesse mit der Verarbeitung bereitgestellter Datensätze betraut, kann aus der Sicht des Entwicklers eines *Broker Task* kein Einfluss auf den parallelen Zugriff der Datenquelle genommen werden. Vielmehr ist während der Entwicklung des verarbeitenden Softwareökosystems auf gegenseitig blockierende Gefahren wie *deadlocks* oder Integrität und Aktualität der Daten bei schreibenden Operationen zu achten. Die transaktionale Kommunikation der Prozesse untereinander ist dabei ebenso zu beachten, wie der programmtechnische Zugriff auf Inhalte einer Datenbank.

Üblicherweise werden Informationen über den Betrieb einer Anwendung innerhalb einer Logfile abgelegt. Diese sind je nach *Log Level* mehr oder weniger aussagekräftig und stellen die Grundlage für das *Status Server* Modul dar. Dieses interpretiert den momentanen Zustand einer Plattform und ist in der Benutzerschnittstelle eines *Broker Task* für die Übertragung von Statusinformationen vorhergesehen. Die Definition eigener REST Pfade innerhalb der Kommunikationsmechanismen von Plattformen ermöglicht eine leichtgewichtige Erweiterung der jeweiligen Instanz um beliebig implementierbare Funktionalität.

Innerhalb des *Dashboard Model* sind alle für einen *Task* definierten Attribute in Tabellen organisiert. Diese Datenstruktur repräsentiert in den meisten Fällen die Relation zwischen einer Menge an eingehenden Daten und spezifischen, die Datensätze verarbeitenden Plattformen. Jede im *Task* verwendete Datenquelle benötigt einen technologiespezifischen Connector, der Metadaten entgegennimmt und der restlichen Programmlogik zur Verfügung stellt. Einzelne Spalten einer Tabelle müssen dabei nicht zwangsläufig aus einer gemeinsamen Datenquelle stammen, sondern können das Resultat von Aggregationen mehrerer Anfragen an unterschiedliche Dienstleister sein. Da die Anwendungslogik des Connectors manuell zu definieren ist, sind an dieser Stelle jegliche Funktionsaufrufe für die Bereitstellung eines bestimmten Attributs anwendbar.

Aktionen können in Abhängigkeit einer Spalte definiert werden und dynamisch per

REST bzw. hinter einem Button abstrahiert oder automatisch anhand definierter Schwellwerte ausgeführt werden. Der Einsatz meherer *Connector*s ermöglicht demnach auch Funktionalität mit aggregierten Parametern. Die auf ein CMS angewandte Aktionen spielen hierbei eine übergeordnete Rolle, weswegen Schlüsselfunktionen wie *create/delete Platform Instance* oder VNC*Connection* auf die jeweilige Virtualisierungstechnologie zu implementieren sind. Während der instantiierung eines *Image*s werden der Plattform entsprungene Metadaten wie ID und IP Addresse innerhalb der *Broker internal DB* persistiert und können in der Tabelle zur Repräsentation von Plattformen verwendet werden. Momentan angewendete Arbeitsschritte einer aktiven Plattform werden durch Kommunikation mit deren *Status Server* in eine *Column* übersetzt. Die Definition von Funktionalität in Abhängigkeit der eingehenden Daten, aber ohne Bezug zu Plattformen oder CMS, beschreibt dabei die Interaktionsmuster des *Broker*s mit Fremdsystemen oder lokalen Mechanismen.

5.3. Aufbau, Platzierung und Kommunikation generischer Komponenten

Mit der Implementierung einzelner Komponenten der in Abbildung 24 aufgezeigten Module entstehen automatisch technologiespezifische Abhängigkeiten. Dabei eingesetzte Bibliotheken und Anwendungsmuster hängen zu einem großen Teil von der verwendeten Programmiersprache ab und können ausschlaggebend für die Bereitstellung von Schnittstellen zu einzelnen Modulen sein. Der generische Einsatz beliebiger Technologien erfordert die konzeptionelle Umsetzung einer dynamischen Architektur. Die folgende Diskussion beinhaltet daher neben Formulierungen grundlegender Abhängigkeiten und Strukturen auch Konzepte der in Abschnitt 4.6 vorgestellten *Apache Thrift* Technologie.

Der Betrieb des *Broker*s und dessen vermittelnde Logik ist, sofern es sich bei der Endanwendung nicht um die Vermittlung lokaler Daten handelt, auf spezifische Strukturen eines *Web Framework*s abgestimmt. Ausgehend von der gewählten Technologie ist neben der Kommunikation mit gewählten CMS Systemen auch die Darstellung und Vermittlung von Datenquellen an eine Verarbeitungsplattform zu implementieren. Die tatsächliche Entwicklung und das *deployment* eines solchen PaaS Mechanismus verlangt tiefes Verständnis über die Aussagekraft der Daten und Kenntnisse über benötigte Architekturen der dynamisch gewählten Verarbeitungstechnologie. Die letztendliche Bereitstellung eines *Cloud Image* in der Hypervisortechnologie eines CSP ist meist mit einmaligem manuellem Integrationsaufwand verbunden und durch einen autorisierten Administrator anzuwenden. Die Anwendung von CMS Funktionen durch den *Broker* verleiht dem Endbenutzer die Möglichkeit von administrativen Aktionen über exklusive Abbilder von konfigurierten Plattformen, deren komplexe Konfiguration durch aufstellen intuitiver Relationen zu spezifischen Datenstrukturen abstrahiert

wurde.

Sprachen-unabhängige Modularisierung

Die bisherige Diskussion untermalt den Einsatz von Vermittlungsschichten als eigenständige Anwendung und isolierte Konzentration der Problemstellung. Je nach Vorhaben kann eine *Broker*struktur jedoch auch nur auf einen kleinen Teilaspekt der Gesamtanwendung bezogen sein und als Zusatzfunktionalität in dieser angeboten werden.

Unter Verwendung der *Apache Thrift* Technologie können Funktionalitäten der *Broker* Struktur innerhalb von Modulen bereits entwickelter Systeme auf programmatischer Ebenen eingebettet werden. Dabei werden alle funktionalen Mechanismen des *Broker*s innerhalb einer zentralen Implementierung gehalten, welche direkt im Anwendungscode referenzierbar sind und von unterschiedlichen Programmiersprachen durch verwenden der jeweiligen Bibliotheken parametrisiert werden können. Für die feingranulare Auftrennung der Anwendungslogik und dezentralisierten Platzierung von Diensten können einzelne Module auf eigenständigen IDLs basieren. Kausalitäten von Aktionen sind zwar statisch im Anwendungscode hinterlegt, jedoch kann das aufgerufene Modul jeder Zeit durch eine neue Version mit identischen Schnittstellen ausgetauscht werden. Diese Vorgehensweise fördert die Flexibilität bei der Technologieauswahl und ermöglicht die entkoppelte Formulierung einzelner Module und deren Versionierung von der Gesamtlogik. Eine Entkopplung von Modulen des *Broker*s ist ebenfalls durch *Message Queue*- und *Streaming* Anwendungen umsetzbar. Dabei entstehen jedoch technologiespezifische Abhängigkeiten, die zu Kompatibilitätsproblemen mit spezifischen Anwendungsmustern führen können.

Abbildung 26 zeigt die Sprachen-unabhängige Modularisierung des Cloud Ressourcen Broker unter Verwendung von *Apache Thrift*. Die in Abschnitt 5.1 aufgezeigten Rahmenbedingungen und Anforderungen können so teilweise voneinander isoliert umgesetzt und durch Kommunikationsmechanismen auf einander abgestimmt werden. Ausgehend von der Problemstellung wird ein Anwendungsmodell eines konkreten Plattform *Broker*s erstellt. Aus diesem Modell können individuelle Aktionen unter Verwendung verschiedener Technologien und Strukturen der Problemdaten formalisiert werden. Dabei verwendete Strukturen beziehen sich auf die in Schaubild 25 aufgezeigten Komponenten und definieren sowohl die eingehende Datenquelle, als auch die zu verarbeitende Plattform.

Unter Anwendung unterschiedlicher *Templates*, können daraus generierte Grundgerüste auf verschiedenen Rahmenwerken basieren. Die in einer *.idl* Datei zusammengefassten Schnittstellen des Systems sind nach einer erneuten Kompilierung durch *Thrift*, in der für die jeweilige Programmiersprache existierende Laufzeitumgebung, anwendbar und kommunikationsfähig. Alle ausgehenden Verbindungen dieser Kom-

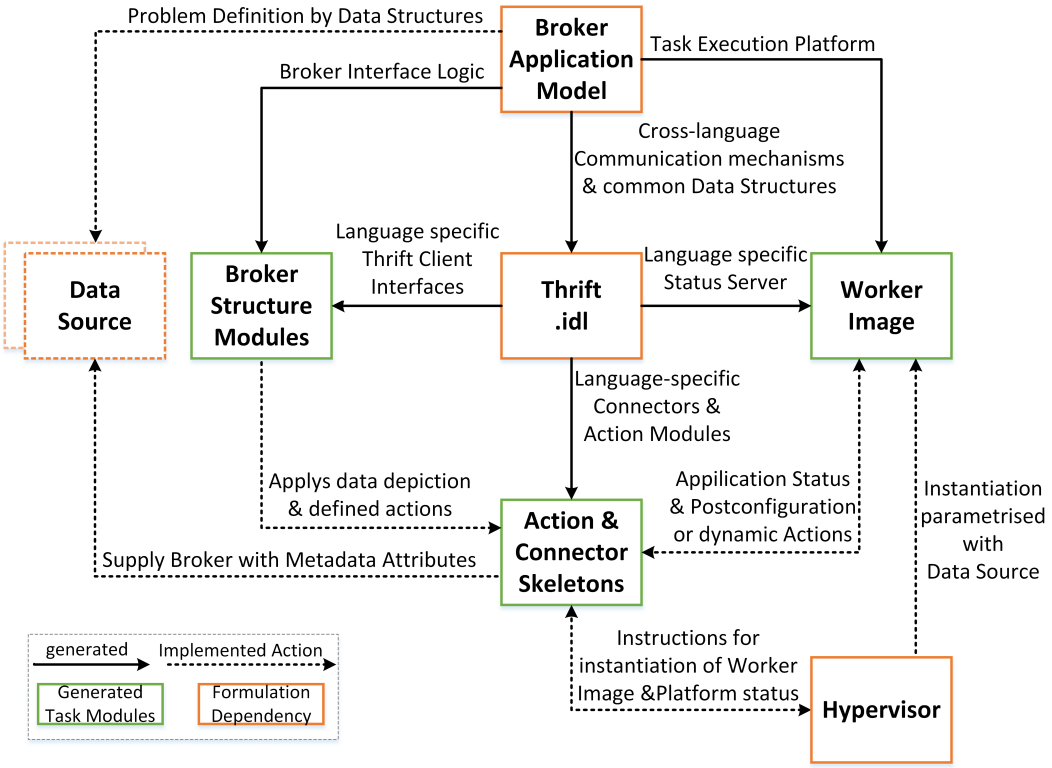

Abbildung 26.: Sprachen-unabhängige Modularisierung des Cloud Ressourcen Broker

ponente zu *Generated Task Modules* sind als Erweiterung des jeweiligen Systemteils zu verstehen.

Die Verwendung eines durch *Thrift* generierten *Language Specific Status Server* ermöglicht die Beschränkung des Kommunikationsaufkommens zwischen Plattform und *Broker* auf diese Technologie. Bei weiteren Verknüpfungen von Anwendungen einer Plattforminstanz zu *Thrift*-basierten Fremdsystemen, können diese Prozesse ebenfalls durch den *Broker* und den darin abstrahierten *Thrift* Aufrufen gesteuert werden.

Die *.idl* Schnittstellen der *Connector*s und Aktionen resultieren in den *Broker Structure Modules* in sprachenspezifische Aufrufen einer entfernten Implementierung. Der serverseitige Anwendungscode kann ebenfalls in allen von *Thrift* unterstützten Programmiersprachen implementiert werden und wird durch den entfernten Aufruf aus der Strukturschicht ausgeführt. Durch die Netzwerkfähigkeit von *Thrift* lassen sich einzelne Module in einer verteilten Umgebung miteinander verknüpfen. Bei rechenintensiven Operationen kann die ausführende Module in physischer Nähe von abhängigen Komponenten, wie Datenquellen oder CMS Maschinen, platziert werden. Die entstehende Dynamik bei der Wahl von Programmiersprachen ermöglicht die entsprechende Implementierung einzelner Module in einer möglichst passenden Sprache. Beispielsweise kann eine Aktion mit tiefgreifenden Telekommunikationsmechanismen verknüpft und innerhalb eines unternehmensinternen *Erlang* Systems als serverseitige Implementation durch den *Broker* aufgerufen werden. In einem solchen Szenario verdichten sich alle

in Abschnitt 5.2 beschriebenen Abhängigkeiten zwischen Komponenten auf logischer Ebene. Bei Anforderungen an Skallierung, Versionierung und physischen Verteilung des *Broker*s ist die durch *Thrift* verursachte zusätzliche Komplexität jedoch ein akzeptabler Nachteil.

5.4. Beispieltechnologien und Implementierungen für Plattform Broker Module

Aus Gründen der Anschaulichkeit wird im folgenden auf eine, auf konkreten Technologien basierende, *Template*-Ausprägung des *Broker*s eingegangen. Jedes softwarebasierte System benötigt eine Laufzeitumgebung, mit Hilfe derer der durch eine Hochsprache abstrahierte Anwendungscode ausführbar gemacht wird. Als Grundlage für die Module des Brokers wird das *Python*-basierte *Django Framework* betrachtet. Mit diesem können neben der eigentlichen Anwendungslogik auch Web-basierte Benutzerschnittstellen implementiert werden. In größeren Projekten ist der Einsatz mehrerer unterschiedlicher Technologien eine gängige Praxis. Jede Anwendung in einem solchen Projekt bedient unterschiedliche Bedürfnisse der Problemstellung und ist einer bestimmten Sprache bzw. Laufzeitumgebung zuzuordnen. Typischerweise interagieren diese Anwendungen über Drittsysteme wie Kafka, *Message Queue*s oder REST Schnittstellen. Unter Verwendung der *Thrift* Technologie können Situationen, in denen bestimmte Eigenentwicklungen unterschiedlicher Projektpartner mit einander kommunizieren, in lokale Funktionsaufrufe mit nativer Parametrisierung der jeweiligen Sprache abstrahiert werden.

Neben unternehmensinternen CMS Systemen können auch cloudbasierte Dienste wie *Amazon EC2* innerhalb des *Cloud Resource Management Module* eingesetzt werden. Die Instantiierung einer Plattform wird im folgenden jedoch mit der CMS Technologie *OpenStack* vorgestellt. Unter Verwendung der in Abschnitt 4.4 vorgeschlagenen Technologien sind die *Cloud Images* zu kompilieren, mit Anwendungen zu bestücken und anschließend in *OpenStack* zu registrieren.

Bereitstellung der Benutzerschnittstelle des Brokers

Für die Bereitstellung einer Benutzerschnittstelle der Vermittlungsschicht sind grundlegend mehrere Technologien und Ansätze denkbar. Eine Web-Applikation ist im Gegensatz zu lokal betriebenen *Client*-Anwendungen des Brokers eine zentral wartbare Lösung. Mit dem *Django* Framework sind für die Problemstellung relevante Eingangsdaten tabellarisch darstellbar. Eine problemlösende Aktion kann darin durch ein auslösendes Ereignis, wie der Betätigung eines *Button*s, mit den Werten einer konkreten Tabellenzeile assoziiert und entsprechend im Anwendungscode umgesetzt werden.

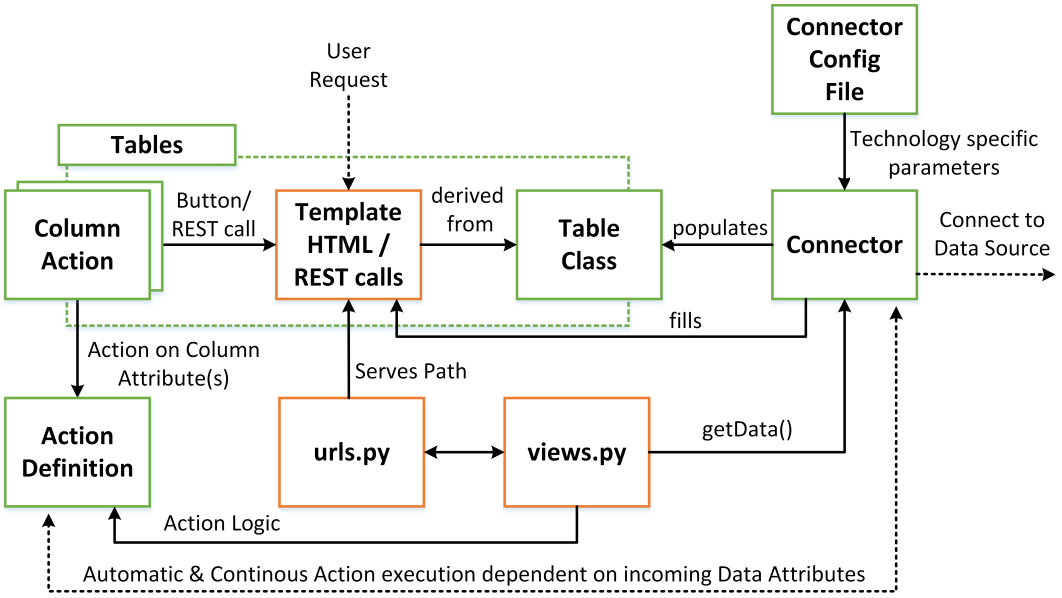

Abbildung 27.: Zusammenhänge eines Broker Task mit Django Modulen

Die in Abbildung 27 dargestellten Komponenten verdeutlichen Zusammenhänge der grün umrandeten *Broker* Modulen und orangenen *Django* spezifischen Eigenschaften. Die im *Task* definierten Metadaten sind in Tabellen organisiert und beruhen auf Klassen. Der *Connector* ist verantwortlich für die Kommunikation mit der Eingehenden Datenquelle und dem Füllen von Objekten der jeweiligen Klasse, welche in einem HTML *Template* dargestellt oder per REST angefragt werden. Die entsprechenden Pfade werden innerhalb der *urls.py* Komponente definiert und mit einer Sicht auf dargebotene Informationen aus der *views.py* verknüpft. Eine *Column Action* repräsentiert die Funktionalität in Abhängigkeit der Attribute einer Tabelle. Die ausformulierte *Action Definition* ist dabei mit der Umsetzung manuell ausgelöster Aufgaben betraut. Das eigentliche *User Request* wird in allen Fällen durch die Implementation des *views.py* Moduls mit externen Datensätzen zusammengeführt. Die Automatische Anwendung von Aktionen, in Abhängigkeit von durch den *Connector* weitergereichten eingehenden Datensätzen, basiert auf zyklischen Abfragen und definierten Schwellwerten.

Das in *OpenStack* bereitgestellte *Horizon Dashboard* basiert ebenfalls auf dem *Django* Rahmenwerk. Mit der Erstellung eines *Horizon Plug-In* lassen sich alle Funktionalitäten der Python API aus *horizon.dashboard.api* verwenden. Im Gegensatz zu einer alleinstehenden *Django* Applikation ist die Verwendung der *OpenStack* REST-API für die Ausführung von Aktionen innerhalb des CMS hierbei nicht notwendig. Im Gegensatz zu REST Schnittstellen, welche sich allgemein seltener ändern, besteht jedoch die Gefahr von veralteten Implementierungen eines *Plug-in* auf Grund von sich ändernden Bibliotheken bei dem Einspielen einer neuen Version des CMS. Wenn alle problemspezifischen Eingangsdaten durch das CMS angezeigt, sinkt außerdem die benutzerseitige *separation of concerns* innerhalb eines *OpenStack Tenant*. Bezogen auf

die in Abbildung 24 definierten Komponenten, werden ebenfalls die Benutzerschnittstelle, Benutzerverwaltung sowie das *Cloud Resource Management Module* von *OpenStack*-spezifischen Mechanismen abhängig gemacht. Wegen dem starken Bezug zu *OpenStack*, verliert ein solcher Broker dabei seine Technologiefreiheit und modulare Eigenschaften.

Eine alleinstehende und unabhängig von dem verwendeten CMS ausführbare *Django* Applikation verlangt, verglichen mit dem *OpenStack* Plug-In, einen Mehraufwand bei der Implementierung. Damit kann jedoch ein höherer Grad der Modularisierung erzielt werden. Die Benutzerverwaltung einer *Django* Applikation basiert auf nach Belieben manipulierbaren und erweiterbaren Modellen. Die in den *views* angebotenen Sichten können mit einer bestimmten Benutzerrolle assoziiert werden und abhängig davon auf unterschiedliche *Templates* verweisen, bzw. diese mit unterschiedlichen Daten füllen. Die jeweiligen Antworten repräsentieren damit auch unterschiedliche Informationen, daraus schlussfolgernde Wissensstände und ausführbare Aktionen. Das Benutzermodell stellet außerdem die grundlegende Referenz der *user accoutnability* dar. Alle durchgeführten Aktionen können so auf einen konkreten Benutzer des Brokers zurückgeführt werden.

Die Implementierung des *Cloud Resource Management Module*s ist die Schnittstelle der von einem Benutzer ausführbaren infrastrukturellen Aktionen. Je nach Unternehmung und Anwendungsgebiet des Brokers, muss ein Benutzer die Anmeldedaten an dem unterliegenden CMS selbst zur Verfügung stellen. Im Bezug auf *OpenStack* kann das *Django* Benutzermodell um ein Feld zur persistierung der Zugangsberechtigung zu dem CMS erweitert werden. Da ein solches *Token* zeitlich begrenzt ist, muss dem Anwender der Vermittlungsschicht eine Möglichkeit zur Neu-Authentifizierung mit *OpenStack* bereitgestellt werden. Mit unterschiedlichen, durch *OpenStack* definierten, Projekten kann dem Benutzer auch eine darauf bezogene Sichtweise auf Eingangsdaten angeboten werden.

Umsetzung von automatisierten Aufgaben der Vermittlungsschicht

Ausgehend von der formulierten Struktur der Eingangsdaten, können beliebige Aktionen in der Programmlogik einer *Django* Applikation mit diesen verknüpft werden. Eine automatische Vermittlung von bestimmten Datensätzen an eine zu instanziierende Ressource ist dabei zwangsläufig auf den jeweiligen *Connector* angewiesen. Mit dem Betrieb einer zusätzlichen und nebenläufigen Komponente des Brokersystems sind diese in Abhängigkeit von definierten Zyklen anzufragen. Die Erstellung einer Verarbeitungsplattform kann dabei von Schwellwerten einzelner Attributen abhängig gemacht werden und die entsprechende *Create Instance* Aktion ausführen.

Solche automatisierten Aktionen eines Systems sind in den seltensten Fällen ohne menschliche Interaktion sinnvoll. Die Definition von Abhängigkeiten kann in abstrakter

Form innerhalb von *Rule Engine*s wie *JESS* formuliert sein. Attribute von eingehenden Datensätzen werden darin interpretiert und je nach Ergebnis mit einem direkten Funktionsaufruf von Modulen der Vermittlungsschicht beantwortet. Ist die Aussagekraft und das Datenaufkommen einer relevanten Datenquelle unbeständig, können automatisch ausgelöste Kommunikationsmuster mit dem CMS in einem hohen Ressourcenverbrauch resultieren. Daher sollte ein für das Projekt zuständiger Administrator über alle durchgeführten Aktivitäten einer solchen Automationskomponente benachrichtigt werden. Bei der Änderung von Datenstrukturen ist dieses Modul eventuell auch von einer grundlegenden Überarbeitung betroffen.

Eigenschaften von erstellten Plattforminstanzen eines Brokers können ebenfalls als Grundlage von automatisierten Aktionen verwendet werden. Dabei wird auf die direkte Kommunikation mit einem *Connector* der Datenquellen verzichtet, vielmehr resultieren Indikatoren über auszuführende Aktionen aus der Broker-internen Datenbank. Ist beispielsweise ein Gültigkeitszeitraum von Plattformen vorhergesehen, können diese durch Kommunikation mit dem CMS eingehalten werden. Die durch den *Status Server* einer VM zurückgegebenen Informationen sind ebenfalls eine mögliche Abhängigkeit von Aktionen. Sind in einer Plattform automatisch durchgeführte Analysen abgeschlossen, können dabei erzielte Ergebnisse durch eine weitere VM verifiziert werden.

5.5. Generierung der Struktur eines Cloud-spezifischen Broker

Die Verwendung einer DSL zur deklarativen Definition eines auf beliebige Datenquellen angepassten *Platform Broker* abstrahiert die bisher geschilderten Zusammenhänge von Modulen. Allgemeingültige Konzepte eines *Broker*s sind dabei in abstrahierter Form als *MetaModel* zusammengefasst. Die Bestimmung der technologiespezifischen Anwendungslogik und konkreten Vorgehensweise zur Konfiguration einer Plattform ist mittels M2C Transformationen definierbar. Die Definition unterschiedlicher Tranformationen ermöglicht die erweiterung von *Broker Templates* um generische Implementierungen.

Konkrete Modelle der Grammatik, welche in der *Extended Backus-Naur Form* deklariert sind, werden momentan von einer Template Engine interpretiert und in Python Module, *Django* Plug-Ins und *Shellscripts* übersetzt. Eine Formulierung von der Erstellung eines Brokers in weiteren Technologien ist nicht ausgeschlossen, aus zeitlichen Gründen konnte jedoch nur auf die oben genannten Beispieltechnologien eingegangen werden.

5.5.1. Struktur der managementseitigen Vermittlungsdefinition

Abbildung 28 beinhaltet die oberste Hierarchie der *Broker* DSL, sowie involvierte Regelsätze für die grundlegende Definition einer Vermittlungsschicht für Plattformen.

Die eine Regel einleitende fett orange umrandete Identifikation einer Regel ist die DSL-interne Bezeichnung der mit dieser Zeile beginnenden Grammatik, welche die Umsetzung der beinhaltenden *ParserRule*s verlangt.

Prinzipiell ist bei der Ausformulierung von Instanzen dieser Grammatik jede Regel mit *ID*s markiert und kann durch diese in weiteren Regelsätzen referenziert werden. Die Benennung von Attributen und Funktionalitäten in generierten Komponenten ist daher zu großen Teilen von den vergebenen *ID*s bzw. für die namensvergabe vorhergesehenen *Features* abhängig. Folgende Rechtecke mit orangenem Rahmen sind in dieser Darstellung als Syntax zu betrachten, wohingegen grüne Komponenten auf die Anwendung einer weiteren Regel verweisen. Die grün gestrichelten Rechtecke repräsentieren eine Verknüpfung zu einer bereits ausformulierten Regel.

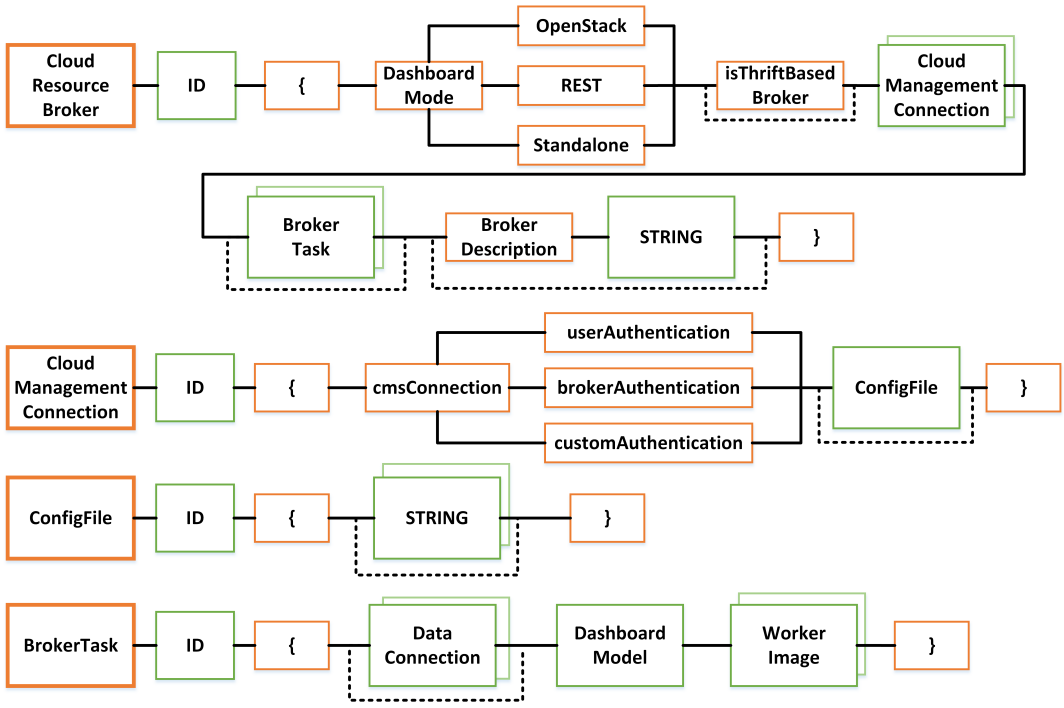

Abbildung 28.: Broker DSL - Grundlegender Grammatikaufbau

Abstimmung auf den Endbenutzer

Ein *CloudResourceBroker* umschließt alle anwendbaren Regeln und beinhaltet die komplette Definition einer Instanz der DSL. Die Auswahl der Bedienung von Vermittlungsschichten ist mittels des *DashboardMode* definierbar und entscheidet über grundlegende Implementierungsmuster der resultierenden Anwendung:

- *standalone*: Die generierte Struktur repräsentiert eine selbstständige *Django* Web-Anwendung, in der alle definierten Tabellen und Aktionen abgebildet werden. Die interne Benutzerverwaltung basiert auf dem *django.contrib.auth.models.User* Modell und ist nach Belieben mit zusätzlichen Attributen anzureichern. Die Be-

dienung von CMS Technologien findet hierbei typischerweise durch das Verwenden einer bereitgestellten REST API statt. Diese Art der Interaktion mit Technologien wie *OpenStack, OpenNebula* oder Diensten wie *Amazon EC2* ist in der Regel eine stabile Schnittstelle für Anwendungsentwickler mit geringem Potential zu veralten.

- OpenStack: Die generierte Struktur entspricht einem *Django* Plugin, welches in das *OpenStack Horizon* Dashboard integriert werden kann. Damit ist dem Entwickler die grundlegende Struktur eines individuellen Dashboards gegeben, welches mit bereitgestellten funktionalen Templates, wie beispielsweise der *Security Groups*, von OpenStack angereichert werden kann. Die Kommunikation mit dem CMS wird durch Python Bibliotheken der Nova API realisiert und ist von der Implementierung der *api.nova.** Funktionen abhängig. Im Gegensatz zu REST basierten Ansätzen ist die Wahrscheinlichkeit der Veraltung von Schnittstellen relativ hoch.

- REST: Die Abbildung der *Broker* Funktionalität in eine Sammlung von REST calls ermöglicht die bequeme Einbindung des Vermittlungsdienstes in automatische Prozesse oder bestehende Anwendungen. Die generierte Struktur repräsentiert das Backend eines *Broker*s in Form einer *Django* Web-Anwendung und ist prinzipiell mit beliebiger Anwendungslogik in Abhängigkeit bereits generierten oder neu definierten Aufrufen der REST Schnittstelle zu erweitern.

Entscheidet sich der Anwendungsmodellierer für einen *Thrift*-gestützten Broker, kann dies mit dem *isThriftBasedBroker Feature* deklariert werden. Die aus der weiteren Formulierung entstehenden Module des Brokers kommunizieren dann über eine *Client-Server* Architektur und können dezentral platziert werden. Die generierte Brokerlogik beinhaltet Aufrufe der *thriftpy* Implementierung.

Die Interaktion mit CMS Technologien ist grundlegend für die Übersetzung von Benutzeranfragen des Brokers in infrastrukturelle Anweisungen. Das aus der *CloudManagementConnection* resultierende Modul ist innerhalb aller *Broker Task*s verwendbar und mit allen in der Grammatik deklarierten Regeln bzw. der Anwendung der Template Engine verknüpft. Da jedes CMS eine individuelle API pflegt, sind Funktionen des generierten CMS *Connector* Moduls manuell mit entsprechender Logik zu versehen. Grundlegende Funktionalitäten sind dabei die Methoden *authenticateWithCMS, createServerInstance* und *deleteServerInstance*, jedoch steht dem Entwickler der Grad an Implementierung und Erweiterung dieses *Connector*s auf die jeweilige Infrastruktur frei. So verlangt die Instanziierung einer Plattform in den meisten Fällen ein *Security Token*, welches dem Anwender unter Angabe von Benutzername und Passwort einen zeitlich begrenzten Schlüssel für Aktionen innerhalb der Hypervisor Infrastruktur ermöglicht. Je nach Einsatzgebiet des *Broker*s können verschiedene Gründe für

die manuelle Authentifizierung eines Benutzers mit dem CMS nach dem Anmeldevorgang an der Vermittlungsschicht sprechen. In diesem Fall von *userAuthentication* reichert die Template Engine das *User Object* mit zusätzlichen Attributen wie *userCmsToken* und *userCmsTokenExpireDate* an. Bei erfolgreichem Anmelden an einem CMS werden diese beiden Attribute eines Benutzers aktualisiert. Für diesen Vorgang wird dem *Broker* eine zusätzliche Repräsentation für die Authentifizierung von *Broker*-Benutzern mit dem unterliegenden Hypervisor in Form von Eingabemasken oder REST Calls bereitgestellt. In überschaubaren Szenarien kann die Komplexität der Vermittlungsschicht durch Verwendung eines einzelnen CMS Account für infrastrukturelle Aktionen unter Angabe des *brokerAuthentication* Schlüsselworts reduziert werden. In großen Unternehmen basiert die Benutzerauthentifizierung meist auf Active Directory (AD) Systemen, welche als Grundlage für das Anmelden am *Broker* verwendet werden können und in einer *customAuthentication* implementierbar ist. Allgemeine Details des Verbindungsaufbaus mit einem CMS wie *hostname, port* oder *tenant* sind in einer *ConfigFile* deklarierbar und können während der Implementierung mit der Python *ConfigParser* Bibliothek in jeglichen Prozess aufgenommen werden.

Integration externer Datenquellen in den Ablauf

Ganz gleich, ob die entstehende *Broker*-Anwendung eine Benutzerschnittstelle oder REST Calls zur Koordination von Aufgaben bereitstellt, werden problemspezifische Eigenschaften innerhalb eines *DashboardModel* festgehalten. Die in Abbildung 29 aufgezeigten Regelsätze sind Bestandteil der in einem *BrokerTask* definierbaren Tabellen.

Die Grammatik erlaubt die Definition mehrerer *BrokerTask*s, welche jeweils auf die Verarbeitung einer bestimmten Datenquelle durch die Erstellung und Konfiguration von *WorkerImage*s abgestimmt sind. Das zugehörige *DashboardModel* repräsentiert die spätere Struktur der *Broker*-seitigen Darstellung von Informationen und Interaktion mit dem CMS durch eine Web-Anwendung oder REST-basierte Aufrufe. Die tabellarische Auflistung von Datensätzen trägt zur intuitiven Umsetzung von Aktionen in Abhängigkeit bestimmter Attribute eines Datensatz bei und ist entweder die Repräsentation von Eingangsdaten oder die Auswirkungen des *Broker*s auf die Infrastruktur. Dabei können einzelne Spalten als *isHiddenColumn* deklariert und nur für die interne Verwendung der Vermittlungsschicht benutzt werden.

- Mit der Verwendung von Referenzen auf *DataConnection* Regeln kann eine Tabelle mit auszuformulierenden Technologien verknüpft werden. Der *Connector* soll in ausformulierter Form in der Lage sein, die Metadaten einer externen Datenquelle bereitzustellen. Daher sind in jedem Fall alle dem Benutzer anzuzeigenden Attribute in Form von *Column*s zu deklarieren. Dementsprechend

ist die Kenntnis über Metadaten und Struktur einer Eingangsdatenquelle unumgänglich und muss durch Aktionen der jeweiligen Technologie spezifiziert werden. Die Beschaffung von deklarierten Metadaten aus einem oder mehreren *Connector*s geschieht durch eine mit dem *Django* Framework verknüpfte *getData()* Funktion, welche für das Füllen von generierten HTML Templates oder JSON Objekten aus den Tabellenobjekten verantwortlich ist.

- Wird eine Tabelle als *instanceTable* deklariert, wird deren Struktur innerhalb der *Broker*-internen relationalen Datenbank erstellt. Diese ist nach momentanem Stand nicht in der DSL Manipulierbar und verlang grundsätzlich eine manuelle Implementierung der darin enthaltenen *connectToData, getData, insertInstance* und *deleteInstance* Funktionen. Alle in dieser Tabelle definierten Spalten sind Aussagen über den Zustand von instanziierten Plattformen, wie deren IP-Addresse oder CMS-internen *Instance ID*. Typischerweise werden Informationen des *Status Server*s eines *WorkerImage* innerhalb der Benutzerschnittstelle dargestellt. Ein Indikator für die Erreichbarkeit von Instanzen ist eine als *representsInstanceSocket* markierte *Column*. Zur tatsächlichen Übertragung von Statusinformationen wird ein, auf diesem Attribut basierender REST Aufruf an den *Status Server* in einer als *isFeedbackColumn* gekennzeichneten Spalte referenziert.

Die Werte der unterschiedlichen Spalten können auf Tabellenebene durch ein *User-Data Feature* für die Konfiguration mit einer Plattform deklariert werden[1]. Darin können beliebig viele *Scripts* enthalten sein, die jeweils einen auszuführenden Befehl darstellen. Innerhalb der mit dem *onStartCommand* assoziierten Zeichenkette können auch Referenzen auf die Spalten der Tabelle verwendet werden. Die in einer *columnRefference* angegebenen IDs der Spalten sind innerhalb des *onStartCommand* jeweils als '§' dargestellt und werden in der angegebenen Reihenfolge durch den *Code Generator* in eine ausführbare Form gebracht. In Kombination mit den in einer Plattform bestehenden statischen Skripten, die nach dem Abschluss des ersten *Boot* Vorgangs ausgeführt werden, sind beispielsweise dafür notwendige Laufzeitvariablen oder Parameter in Konfigurationsdateien deklarierbar.

Die Formulierung von einem Benutzer angebotenen Aktionen findet ebenfalls auf Tabellenebene statt. Dabei ist eine Auswahl Essenzieller Funktionalität mit dem CMS möglich und kann auf bestimmte *Column*s angewandt werden. Die hinter einem Button abstrahierte Anwendungslogik kann während der Implementation mit beliebig zusätzlichen Mechanismen angereichert werden. Die Definition der *custom* Aktion rechtfertigt sich durch die individuelle Anreicherung des *Broker*s mit unternehmensin-

[1] Die Benennung dieses Features ist auf den in *OpenStack* für die Angabe eines Startskripts bei der Instanziierung einer VM angebotenen *user_data* Parameter abgeleitet.

terner Funktionalitäten und Prozessen, die keine CMS-relevanten Abhängigkeiten besitzen, sich jedoch auf eine bestimmte Zeile bzw. Spalte beziehen. Beispielsweise kann die Aussagekraft eines Datensatzes durch Aggregation mehrerer *DataConnection* innerhalb des *Broker*s gesteigert und in konzentrierter Form an unternehmensweite Managementtechnologien versendet werden.

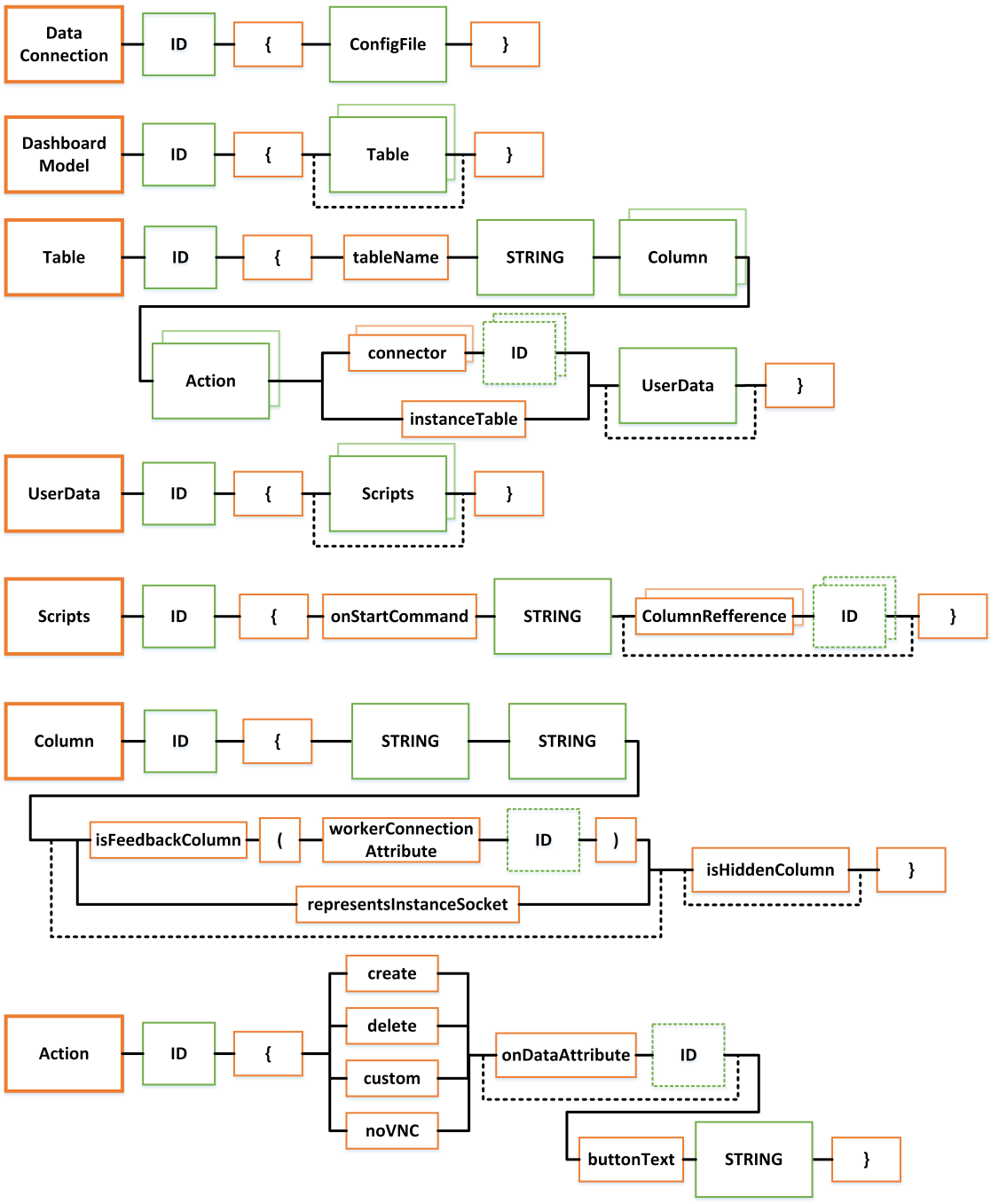

Abbildung 29.: Broker DSL - Definition der Represantion von Metadaten und Aktionen

5.5.2. Strukturierung und Konfigurationsmechanismen der Verarbeitungsplattform

Die Definition einer verarbeitenden Plattform ist mit der *WorkerImage* Regel umzusetzen. Wie in Abbildung 30 dargestellt, handelt es sich dabei um eine verschachtelte Struktur. Das *WorkerImage* basiert auf dem Command Line Interface (CLI)-Werkzeug

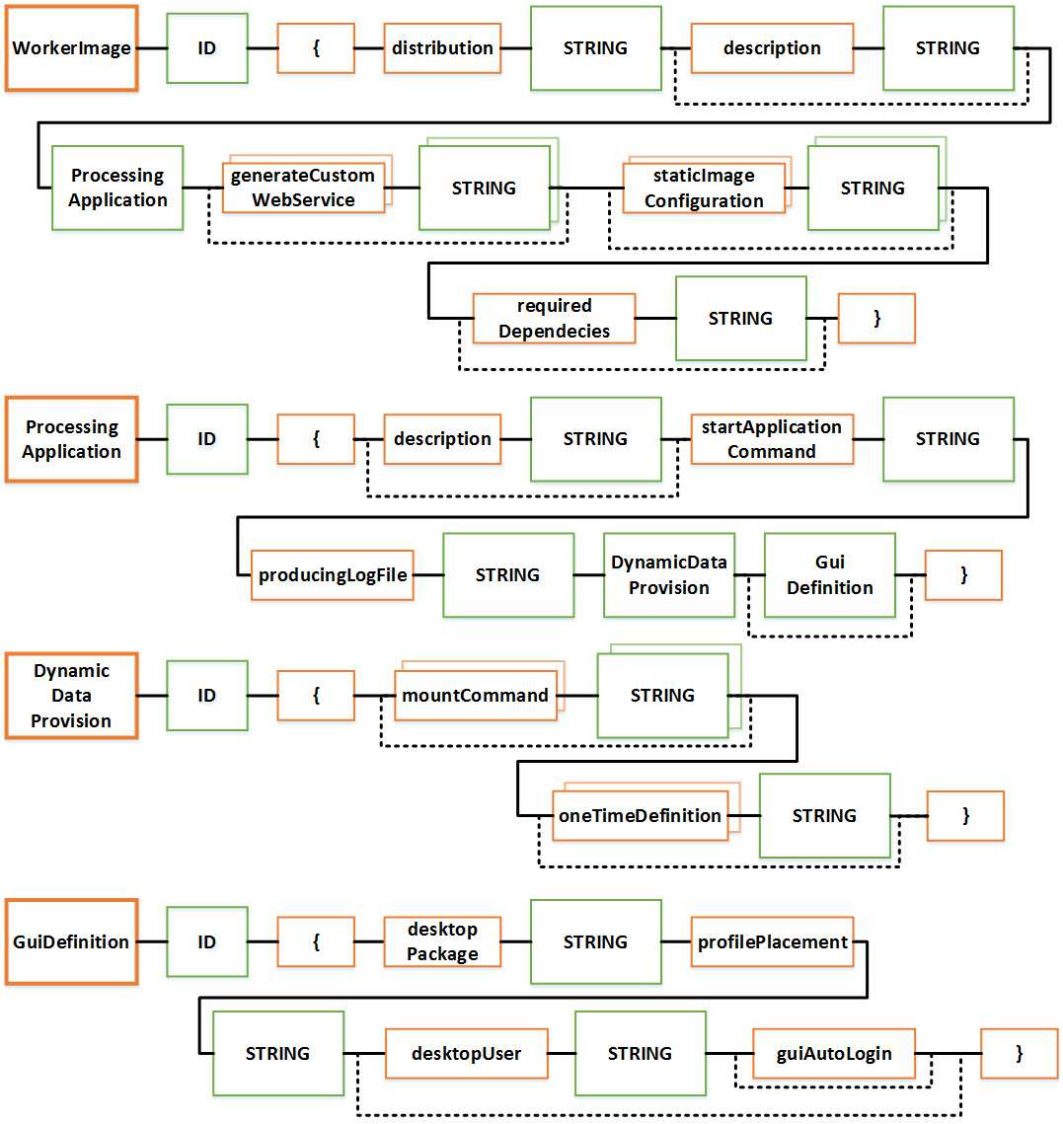

Abbildung 30.: Broker DSL - Definition der Verarbeitungsplattform und Konfigurationsmechanismen

virt-builder(von libguestfs), welches innerhalb eines *Shellscript*s mit Parametern bestückt wird und diverse Distributionen von Linux Derivaten unterstützt. Unter Verwendung der *staticImageConfiguration* Regel ist die folgende Zeichenkette für die grundlegende Anreicherung des *Cloud Image* mit statischen Konfigurationen gedacht, welche jeweils in *–run-command* Parameter von *virt-builder* übersetzt werden. Hierbei kann ebenfalls die manuelle Installation der Software von Drittanbietern umgesetzt werden. Alle in *staticImageConfiguration* formulierten Skripte werden durch *virt-builder*, sofern

nicht explizit angegeben, als Superuser im resultierenden *Image* ausgeführt. Meist ist der erfolgreiche Betrieb einer Plattform an Systempakete gekoppelt, welche innerhalb der *requiredDependencies* anzugeben sind. Abhängig von der gewählten Distribution kann das *WorkerImage* unter Verwendung des jeweiligen Paketmanagers mit dem –*install* Parameter angereichert werden. Für die Definition zusätzlicher REST Calls des Status Servers ist die *generateCustomRestService* Regel anzuwenden und an entsprechender Stelle zu implementieren. Dieses einmalige "Branding" des *WorkerImage* ist damit für alle Instanzen gültig.

Eine *ProcessingApplication* ist als problemspezifische Software zur Verarbeitung von Informationen zu betrachten. Mit dem *startApplicationCommand* Ausdruck ist die Art und Weise der automatischen Ausführung der Anwendung zu definieren. Abhängig von dem Integrationsgrad der Software in das OS kann beispielsweise die */etc/rc.local* Datei einer CLI-basierten Plattform mit dem Aufruf verschiedener Anwendungen bestückt werden. In diesem konkreten Fall sind blockierende Anwendungen innerhalb des *startApplicationCommand* in Konstruktionen, wie *nohup* Befehle oder der Ausführung in *screen* Umgebungen, einzubetten. Das definierte Kommando wird durch die Verwendung von CLI Werkzeugen wie *sed* in *–run-command* Parametern von *virt-builder* übersetzt und in den Startprozess einer Instanz eingepflegt. In verschiedenen Anwendungsfällen ist die Verarbeitungssoftware weder CLI-basiert noch durch ein Web-Interface zu bedienen, sondern verlangt die direkte Interaktion innerhalb der grafischen Oberfläche des virtualisierten Servers. Dieser Umgang mit der Plattform ist mit dem optionalen *requiresGUI feature* auszudrücken, welches das resultierende *Image* mit GUI-Benutzern und Profilen für die Konfiguration und Bedienung anreichert. Die standartmäßige Definition der Oberfläche ist ein Lightweight X Desktop Environment (LXDE) mit automatischem Anmelden des *[WorkerImage]_applicationUser* und direktem Starten von betroffenen Anwendungen durch in *~/.config/autostart/[applicationName].desktop* platzierten Profilen. Die in diesen Dateien deklarierten Befehle und Strukturen sind ein Abbild der Deklaration des *startApplicationCommand features*.

Die Angabe einer *producingLogFile* wird innerhalb des Status Servers zum Propagieren von Interpretationen des momentanen Verhaltens einer Instanz an die Vermittlungsschicht verwendet. Wegen der Bereitstellung von REST Schnittstellen ist eine Einbindung des Status Server in weitere Anwendungen möglich. So ist beispielsweise eine unternehmensweite Managementsoftware in der Lage, auch den Zustand von Plattform Instanzen mit in den Verwaltungsprozess aufzunehmen.

Um die Verknüpfung von Eingangsdaten mit der Anwendungssoftware zu gewährleisten wird sich der *DynamicDataProvision* Regel bedient. Der in *mountCommand* definierte Befehl wird in zeitlicher Reihenfolge vor dem *startApplicationCommand* bei jedem Starten der Plattform ausgeführt. Da der Betrieb der Plattform ohne die Anga-

be von zu verarbeitende Abhängigkeiten in einer manuellen Konfiguration resultiert, ist die Deklaration dieses *feature*s meist ein essentieller Bestandteil der Definition von *Broker*n dieser Art. *Mount*-Befehle von dabei verwendeten Technologien wie NFS, Secure SHell File System (SSHFS) oder HDFS bestehen meist aus Referenzen zu eingehenden Metadaten und werden daher dynamisch während der Erstellung einer VM in Form von *user_data* Skripten übergeben, welche die Manipulation von Profilen eines Benutzers bzw. das Hinzufügen von Kommandos in den Initiierungsprozess der jeweiligen Maschine tätigen. Ist eine verarbeitende Anwendung beispielsweise mit einem Parameter für den Pfad zu einem vorab bereitgestellten Datensatz zu starten, kann dieser als Umgebungsvariable des Benutzers exportiert und innerhalb des *startApplicationCommand feature*s verwendet werden.

Möglicherweise ist die Plattform durch eine Sammlung an mit statischen Attributen zu füllenden Konfigurationsdateien an jeweilige Datensätze anzupassen und anschließend in der Lage, den *Mount*vorgang in Eigenregie zu ubernehmen. Die vom *oneTimeDefinition* Ausdruck gefolgte Zeichenkette repräsentiert das bei der Instanziierung einer Plattform angewandte *Shellscript* und beinhaltet meist das Füllen von Konfigurationsdateien mit von den Eingangsdaten abhängigen Werten. Dieses *feature* bietet dem Entwickler einer Plattform ebenfalls die Möglichkeit von einmalig auszuführenden Systembefehlen innerhalb einer Plattform mit dynamischen Parametern.

Tabelle 1.: Übersicht der Grammatikregeln

Rule	Applied Features	Nested Relations
BrokerDSL	Descides the *DashboardMode* and may contain a description. Also defines, if *Thrift* should be used for containing elements-	Is root element of DSL; 1..n CloudManagement Connection; 0..n BrokerTask
Cloud Management Connection	Defines the type of user authentication with CMS	0..1 ConfigFile
Broker Task	Container for input data processing brokerage tasks	1..1 DashboardModel; 1..n WorkerImage; 0..n DataConnection
Data Connection	Represents connector to data source (Any configurable technology specific parameters should be placed here)	1..1 ConfigFile
Dashboard Model	Wrapper for metadata	0..n Tables

Table	Represents broker logic definition, displays values in dashboard and is either used for incoming data or processing platforms	0..n Column; 0..n Action; 0..1 refferences to Data Connection
Column	Represents an attributes of broker structure and my be defined as platform status connection to the dashboard. also may be hidden	-
Action	A broker action for custom or CMS functionality as Button or REST call	refferences 0..1 Column
UserData	Represents actions to be performed on instantiation of platforms	0..n Scripts
Scripts	Represents a line in configuration script of platforms and may be connected with a Column	1 STRING; refferences 0..n Column
Worker Image	The executing VM image for processing of data. Is based on an Operating System, system package dependencies, static configuration and custom REST calls for interaction with instances.	1..1 ProcessingApplication; 1..1 DynamicDataProvision; 0..1 GuiDefinition
Processing Application	The actual platform enabling software must is started by individual commands and produces logfiles	1..1 DynamicDataProvision; 0..1 GuiDefinition
Dynamic Data Provision	Definition of mount commands and populating of log files related to incoming data sources	-
Gui Definition	Enables the declaration of Desktop technologies, related profile placement and provides the definition of a Desktop user as well as its auto login	-
ConfigFile	Simple configuration file representation declaring keys (values must be set in resulting application)	0..n STRING

Die mit der Code Generierung entstehenden Module repräsentieren eine Instanziierung von der durch die DSL abstrahierten Architektur. Die Komplexitätsanalyse eines solchen Applikationsgerüsts ist erst mit der Formulierung von individuellen Programmteilen aussagekräftig. Bei der Implementierung eines Templates können jedoch

möglichst passende Strukturen vorgegeben werden, welche die Anforderungen an die Komplexität des Brokersystems auf Quellcodeebene berücksichtigt. Die Komplexitätsanalyse einer Anwendung beschäftigt sich primär mit darin verwendeten Datenstrukturen und Algorithmen [MJ08]. Daher beeinflussen alle darin involvierten Technologien maßgeblich die aus der Komplexität resultierende Performanz der Anwendung.

5.6. Zusammenfassung

In dem vorliegenden Kapitel wurde eine DSL zur Definition von Brokerarchitekturen um Bezug auf zu eingehenden Datenquellen und deren Verarbeitung durch virtualisierte Plattformen vorgestellt. Um ein tieferes Verständnis für die Umsetzung und den Betrieb einer derartigen Vermittlungsschicht zu bilden, werden im folgenden die darin involvierten Aktoren vorgestellt. Die in Abbildung 31 gezeigten Module repräsentieren die vorgestellten Komponenten des Brokers. Ausgehend von einer Datenquelle wird ein

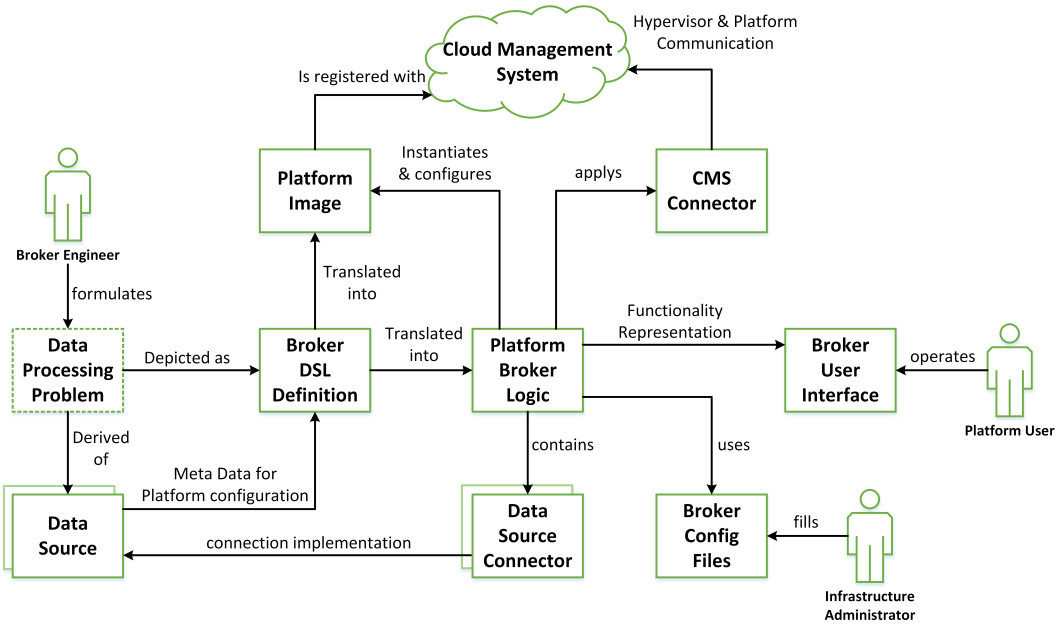

Abbildung 31.: Definition von Plattform Brokern und involvierte Aktoren

Verarbeitungsproblem abgeleitet und in einer Instanz, der in Abschnitt 5.5 vorgestellten Broker-DSL durch einen *Broker Engineer*, formuliert. Diese beinhaltet mindestens die für die Problemstellung notwendigen Metadatenstruktur für die Konfiguration von Plattformen. Die DSL wird in unterschiedliche Skripte zur Erstellung der *Platform Image*s und Datenstrukturen der *Platform Broker* Logik übersetzt. Die in der Logik enthaltenen Kommunikationsmuster mit einer Datenquelle sind innerhalb des *Data Source Connector* zu implementieren. Mit einem *CMS Connector* werden in der DSL definierte Kommunikationsmuster mit CMS Systemen abgebildet und durch die Brokerlogik angewandt. Formulierte Aktionen über die Datenquelle werden durch diese

in die Instanziierung eines damit konfigurierten *Platform Image* übersetzt. Infrastrukturelle Begebenheiten der Brokerumgebung, wie beispielsweise die IP-Adresse eines verwendeten CMS, sind durch den *Infrastructure Administrator* innerhalb von *Broker Config Files* bekannt zu geben. Die Funktionalität dieser Vermittlungsschicht wird dem *Platform User* in einem *Broker User Interface* bereitgestellt. Das Hauptaugenmerk dieser Architektur liegt auf der Effizienz der Erstellung von komplexen Vermittlungssystemen. Mit der DSL umsetzbare Problemstellungen sind abstrakt definierbar und können von Anwendungsentwicklern an vorgegebener Stelle mit Programmlogik versehen werden. Die Einhaltung von zugewiesenen Schnittstellen der Module ermöglicht das manuelle wechseln von Technologien der einzelnen Komponenten. Die Bereitstellung einer Benutzeroberfläche soll dabei die Komplexität der Aufgabenstellung verringern und den Anwender durch automatisierte Prozesse in seiner Unternehmung unterstützen.

6. Szenarien

Im folgenden wird die in Kapitel 5 beschriebene Architektur an mehreren Anwendungsbeispielen demonstriert. Mit dem breiten Anwendungsspektrum der Vermittlungsschicht soll der generische Aspekt dieser Arbeit unterstrichen werden. Dabei wird die vorgeschlagene DSL im Bezug auf ein einführendes Szenario angewandt. Genannte Details der Implementierung basieren auf den, in Kapitel 4, genannten Technologien.

6.1. Showcase

Das folgende Szenario bezieht sich auf die Verbindung zweier Datenquellen zu einer Vermittlungsschicht und der Bearbeitung anhand der darin enthaltenen Metadaten. Dabei ist die Arbeitsweise der Plattformen, sowie Details der Kommunikation von Modulen für eine anschauliche Beschreibung bewusst abstrakt gehalten. Anstatt eine konkrete Problemstellung zu lösen, bezieht sich dieses Anwendungsbeispiel auf grundlegende Zusammenhänge, Abläufe und Strukturen der Brokerarchitektur. Für die detailierte Implementierung dieses Szenarios wird auf Anhang A verwiesen.

Problemstellung

In einem *standalone* Plattform Broker sollen Metadaten repräsentierende Datensätze beider Speichertechnologien für eine weitere Verarbeitung bereitgestellt werden. Die in jeweils einer Tabelle dargestellten Metadaten der Datenquelle sind jeweils einem separaten Plattformtyp zuzuordnen und benötigen individuelle Konfigurationsmechanismen. In einer weiteren Tabelle soll eine aggregierte Sichtweise auf beide Informationsquellen eine neue Analysegrundlage bilden und resultiert daher in einem eigenen *Platform Image*. In einer vierten Tabelle werden dabei die momentan instanziierten Plattformen gemeinsam angezeigt. Für den direkten Verbindungsaufbau mit der jeweiligen Instanz wird eine *noVNC* Verbindung durch einen *Button* abstrahiert. Eine weitere Aktion ist für den Beginn einer Prozesskette verantwortlich, in der ein *Snapshot* der jeweiligen Instanz durch OpenStack erstellt und anschließend auf separaten Datenträgern weiter verarbeitet wird. Des weiteren ist der momentane Status einer Plattform aus der Tabelle von Instanzen ablesbar. Die in Abbildung 32 dargestellten Zusammenhänge sollen für die unterschiedlichen Arbeitsschritte zur Erstellung dieser lauffähigen Vermittlungsschicht mit mehreren Datenquellen sensibilisieren. Da alle Aktionen mit dem *Apache Thrift* Framework ausgeführt werden sollen, verwenden

die generierten Strukturen die Client-seitige Implementierung der auf die IDL aufbauenden Funktionalitäten. Der Vollständigkeit halber sind in dieser Abbildung auch die generierten Strukturen der Vermittlungsschicht ohne das *isThriftBasedBroker* Schlüsselwort aufgezeigt.

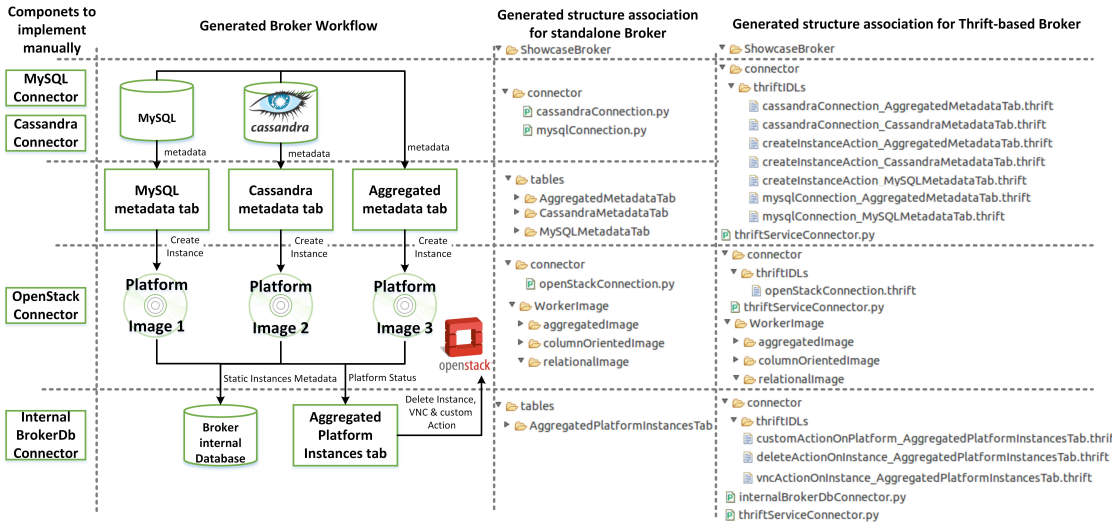

Abbildung 32.: Anwendungsbeispiel eines Plattform Broker mit mehreren Datenquellen

Die Implementierung der Verbindung zur relationale *MySQL*- und spaltenorientierte *Cassandra*- Technologien ist die grundlegende Voraussetzung für die Arbeitsweise des Brokers. Eine Ausformulierung der in den *cassandraConnection.py* und *mysqlConnection.py* Module generierten Grundgerüste zum Verbindungsaufbau und der Erfragung von Datensätzen ist dabei für eine *standalone* Anwendung unumgänglich. Mit den technologiespezifischen Implementierungen als *Thrift Server* können diese Komponenten 'nahe' bei der jeweiligen Ressource platziert werden. Die an die Konnektoren zurückgegeben Liste an Objekten repräsentieren die in der DSL definierte Tabellenstruktur und können auf Grund des vermiedenen Netzwerkverkehrs bei der Anfrage an die Datenquelle auch sensible Daten in einer Vorverarbeitung mit einbeziehen.

Die generierten HTML Templates und *Django* Strukturen sind bereits mit dieser Verarbeitungslogik verknüpft und stellen daher den statischen Anwendungscode dar, der in Abhängigkeit von Benutzereingaben auf individuelle *Thrift* Dienste angewandt wird. Da in diesem Beispiel eine *standalone* Variante verwendet wurde, sind alle Operationen der Vermittlungsschicht, inklusive der Datenanfrage, mit einer vorherigen Authentifizierung des Benutzers verbunden.

Die Virtualisierung von Plattformen stützt sich auf OpenStack. Durch das in der DSL deklarierte *userAuthentication* der *cmsConnection* wird der Authentifizierungsvorgang mit dem CMS manuell in einer Eingabemaske der Vermittlungsschicht durchgeführt. Das resultierende *Token* berechtigt den Anwender für ein bestimmtes Zeitfenster zu administrativen Operationen auf der Infrastruktur des zugehörigen *Tenant*

mittles der *Nova-Compute API*.

Die Erstellung der auf OpenStack angepassten *QCOW2 Cloud Images* wird für jeden Plattformtyp separat in einem eigenen Skript definiert. Dabei agieren *Platform Image 1* und *2* vollautomatisch und benötigen auf Grund angebotener REST Schnittstellen keine grafische Oberfläche auf Betriebssystemebene. Bei der Bearbeitungen von Daten beider Datenquellen kommt ein grafisches Werkzeug zum Einsatz, weswegen *Platform Image 3* mit einem LXDE Paket, sowie einem speziellen Benutzer für die Ausführung der Anwendung angereichert wird. Das Analysewerkzeug wird durch Profile des LXDE automatisch beim Hochfahren einer Instanz ausgeführt und steht dem Benutzer dank automatischem Anmelden an der Benutzeroberfläche direkt zur Verfügung.

Mit der Registrierung dieser *Cloud Image*s in OpenStack ist die resultierende *Image ID* innerhalb der zutreffenden *Connector*-Konfigurationsdatei einzutragen, die bei entsprechenden Instanziierungen verwendet wird. Ebenso sind Angaben zu Ressourcen der entstehenden VM oder Netzwerkumgebungen in dieser definierbar und einsehbar. Die Definition von statischen Konfigurationsparametern einer Plattform kann ebenfalls an dieser Stelle stattfinden. Die *Create Instance* Aktionen der Metadatentabellen sind daher in separaten *Thrift Services* zu formulieren, die innerhalb des *OpenStack-Connector.py* Brokermoduls als entfernter Funktionsaufruf angefragt werden.

Diskussion

Mit der Platzierung von *Connector*-Implementierungen an der jeweiligen Ressource wird eine Separierung der verschiedenen Anliegen auch physisch umgesetzt und unter Umständen so die Netzwerklast verringert. Sind die Technologien der Datenquelle, sowie die Vermittlungsschicht durch ein und das selbe CMS bereitgestellt, verkürzen sich die Wege der Informationsübertragung, da über ein privates Netzwerk kommuniziert werden kann. Werden nicht nur Metadaten, sondern auch die referenzierten Rohdaten auf Maschinen in diesem Netzwerk platziert, verringert die Instanziierung der Verarbeitungsplattformen durch das gleiche CMS die Distanz der Datenübertragung erneut. Eine Umsetzung des geschilderten Anwendungsfalls durch physische Maschinen, resultiert auf Grund des Kommunikationsbedarfs der auf verteilten Ressourcen platzierten Informationen, in Performanceeinbußen. Mit der Isolation von problemspezifischen Verarbeitungsplattformen, die einer gemeinsamen Domäne zuzuordnen sind, ist eine granulare Strukturierung von beispielsweise unterschiedlichen, in ein Projekt involvierten Abteilungen einer Organisation umsetzbar.

Eine durch den Broker instanziierte Ressource ist in den seltensten Fällen das letzte Element einer Prozesskette. Erzeugen diese Plattformen wiederum neue Erkenntnisse, die in einem weiteren Arbeitsschritt durch ein Kollektiv weiterverarbeitet werden, ist die Zieldatenhaltung dementsprechend zu formulieren. Neben der mehrfachen Bearbeitung der in die Plattformen eingehenden Datensätze durch eine einzelne VM, ist

die Verwaltung der mit dieser Datenquelle mehrfach instanziierten VMs zu beachten. Bei einer großen Menge an instanziierten Plattformen, führt deren Auflistung in einer gemeinsamen Tabelle schnell zu unübersichtlichen und möglicherweise verwirrenden Darstellung. Die Gruppierung nach Plattformtyp oder IP-Adressbereich von Instanzen verringert dieses Problem nur oberflächlich. Die *Status Server* der unterschiedlichen *Platform Image*s tragen ebenfalls nur zu kleinen Teilen zur Identifikation von VMs bei.

6.2. Analyse von Daten im Bereich Autonomes Fahren

Wie in Kapitel 2 bereits angedeutet, entstehen bei der Entwicklung von neuartigen Technologien große Mengen an Daten. Mit steigender Anzahl verbauter Sensoren in einem Prototyp ergibt sich auch eine höhere Informationsdichte.

Problemstellung

Kaum ein anderes Thema der modernen IT wurde in den letzten Jahren so öffentlich diskutiert, wie der Einsatz von autonomen Fahrzeugen. Derartige Systeme befinden sich zu dem Zeitpunkt dieser Arbeit noch in der Entwicklungsphase. Neben der für die autonome Fahrt benötigten Rechenleistung, ergeben sich durch verschiedenste Radarsysteme, Video- und Audioaufzeichnungen oder Wärmesensoren in kürzester Zeit riesige Datenmengen. Bei Auffälligkeiten oder Fehlverhalten des Fahrzeugs in autonomen Fahrmanövern wird unter Umständen auf eine manuelle Analyse der gefahrenen Teststrecke zurückgegriffen. Eine auf diese Visualisierung angepasste Analysesoftware kann in einem solchen Fall auch in virtualisierten Plattformen betrieben werden und die Ressourcen der lokalen Maschine eines Analysten entlasten. Abbildung 33 zeigt den Anwendungsfall einer Vermittlungsschicht für die Bereitstellung einer virtualisierten Analyseplattform von Daten autonomer Fahrzeuge. Dieser spezielle Broker ist dabei nicht zuletzt aus Sicherheitsgründen in der *Private Cloud* einer Organisation anzusiedeln.

In einer *Vehicle Data* Tabelle sind die Metadaten aller Testfahrten aufgelistet. Neben den Anfangs- und Endkoordinaten der Teststrecke befindet sich in dieser Datenhaltung auch die Referenz auf die tatsächlich gesammelten Daten. Auf Grund des gigantischen Speicherverbrauchs solcher Aufzeichnungen ist die Persistierung in einer auf die Datenstrukturen angepassten Technologie zu bewerkstelligen und gegebenenfalls vor der Analyse aufzubereiten. Mit einem *Create Analysis Platform Button* ist dem *Vehicle Log Analyst* die Möglichkeit der Erstellung einer Plattform gegeben, die automatisch mit der jeweiligen Aufzeichnung konfiguriert gestartet wird. Um die Daten innerhalb der Instanz zur weiteren Verarbeitung verfügbar zu machen, muss die Plattform prinzipiell mit der jeweilige Technologie kommunizieren können.

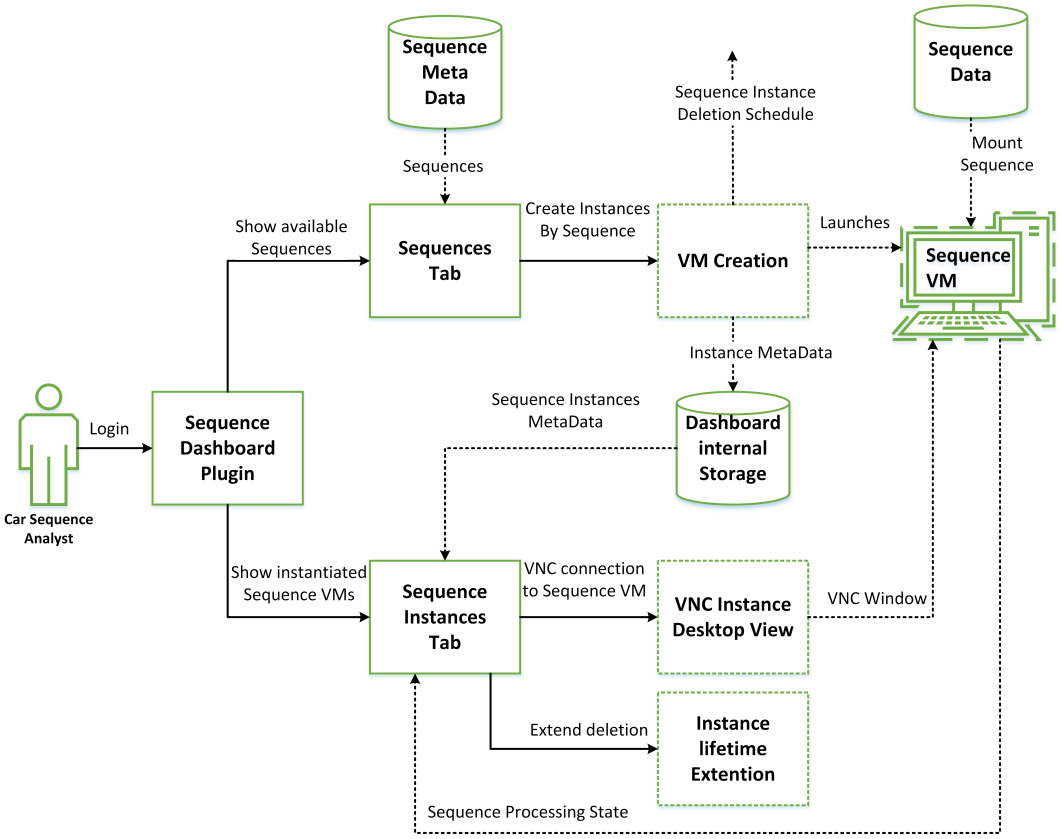

Abbildung 33.: Brokermechanismus für die Datenanalyse von autonomen Fahrzeugen

Die Analyse einer aufgezeichneten autonomen Fahrt ist in den meisten Fällen nur in einem bestimmten Zeitrahmen von Interesse. Beispielsweise können sich mit festgelegten Meilensteinen auch möglicherweise Strukturen des Datenformats oder der Analysesoftware selbst ändern. Sind derartige Zeitpunkte im Vorhinein bekannt, kann die Vermittlungsschicht um eine *Instance Deletion Schedule* erweitert werden. Dabei werden alle aktiven Instanzen in definierbaren Abständen auf einen gültigen Zeitrahmen oder dem Überschreiten eines bestimmten Datums getestet und im Negativfall aus dem CMS entfernt. Selbstverständlich ist dabei die Broker-interne Abbildung von instanziierten VMs mit den tatsächlich durch das CMS bereitgestellten Plattformen abzugleichen. Die Entlastung der Virtualisierungsinfrastruktur hat neben der Zusicherung von freien Ressourcen zu einem bestimmten Zeitpunkt auch abschreckende Wirkung auf die bewusste Zweckentfremdung von VMs durch Benutzer.

Da es sich bei der Anwendung von Analyseprogrammen einer Plattform um manuell auszuführende Operationen handelt, zeigt der Status einer VM in der *Vehicle Platform Instances* Tabelle lediglich deren Erreichbarkeit oder letzte Programmaktivität aus einer Logdatei an. Auf Grund der vorgesehenen Aktivitätsspanne einer Plattform ist auch deren *Deletion Date* für den Anwender des Brokers von Interesse. Neben der *noVNC* Verbindung zu einer Plattform kann deren Lebensspanne ebenfalls durch einen *Button* verlängert werden. In einem solchen Fall ist die betreffende Zeile der im *In-*

stance Deletion Schedule Modul verwendeten Datenbank abzuändern. Das Freigeben von virtualisierten Ressourcen geschieht in diesem Anwendungsfall ausschließlich über diese Komponente, eine entsprechende manuelle Aktion ist jedoch als zusätzlicher *Button* denkbar.

Diskussion

Erstrecken sich derartige Unternehmungen über mehrere Kontinente, kann mit dem Zusammenschluss von *Private Cloud*s einzelner Filialen in einem gemeinsamen Archiv relevanter Aufzeichnungen kooperiert werden. Die Isolation von Plattformen und Vergabe von Zugriffsrechten der Mitarbeiter über die physischen Virtualisierungsumgebungen begünstigt neben *Remote Reviews* auch die Archivierung von besonders interessanten Aufzeichnungen. Sind gesammelte Daten bei der bisherigen Analyse auf die lokale Maschine eines Analysten zu kopieren, stellt die 'nahe' Platzierung von VMs an der Datenquelle und der entfernten Zugriff auf diese Plattformen eine performante Alternative dar.

6.3. Broker für verteilte Plattformen im Umfeld von Big Data Analysen

Die Verarbeitung von gigantischen Mengen an Datensätzen ist im Zeitalter von *Big Data* nicht mit traditionellen RDBMS zu bewältigen. Eine Transformation vieler unterschiedlicher Datenstrukturen kann, unter Verwendung von No-Structured Query Language (SQL) Technologien performant in verteilten Umgebungen realisiert werden [ER16]. Typischerweise beinhaltet die Architektur von System, wie *HBase* eine zentrale Managementkomponente, welche die Konfiguration und Registrierungsstelle der verarbeitenden und datenhaltenden Instanzen darstellt.

Die Funktionalität einer Anwendung verschiebt sich in einem solchen Anwendungsfall von den *Cloud Images* auf den Quellcode, welcher durch den Anwendungsentwickler einer Plattform bereitgestellt wurde. Beispielsweise wird eine *Apache Spark* Anwendung an zentraler Stelle in Auftrag gegeben und durch das Cluster bearbeitet. Der Entwickler hat dabei Kenntnis über erwartete Datenvolumen und der verbundene Ressourcenlast des Clusters. Verarbeitungsmuster der jeweiligen Technologien können in Plattformprofile wie *Batch* oder *Stream Processing* abstrahiert werden und mit Standardeigenschaften, wie der Anzahl verschiedener Module einer Plattform angereichert werden. Die horizontale Skalierung einer Infrastruktur soll die Reaktion auf dynamische Lasten von Anwendungen ermöglichen. Ein auf die jeweilige Technologie abgestimmter *Broker* ist daher neben der Koordination der Instanzen ebenfalls mit der Konfiguration von allen an der Technologie beteiligten Ressourcen betraut.

Problemstellung

Abbildung 34 verdeutlicht die Platzierung eines *Broker*s zur Umsetzung des Zusammenspiels von verteilten und dynamisch skalierbaren Infrastrukturen. Die verwendete Technologie eines verteilten Systems wie *Apache Spark* oder *Cassandra* kann nicht in jedem Szenario ohne einen Neustart der Plattform umgesetzt werden. Alle verteilbaren Module sind jedoch durch vorkonfigurierte *Cloud Images* instanziierbar und werden durch den *Broker* während der Kommunikation mit dem Hypervisor in dessen interner Datenbank registriert. In diesem lokalen Register von dynamischen Plattformkomponenten wird ebenfalls der Status von *Messaging Node* Instanzen bereitgestellt. Ist die initiale VM Konfiguration einer *Worker Node* abgeschlossen, assoziiert sich diese durch Anfragen an die *Broker*-DB mit einem zugehörigen Messaging Server. Auf der Gegenseite muss die *Worker Node* innerhalb der *Messaging Node* registriert werden, was durch SSH Verbindungen oder eigenentwickelte Serverschnittstellen in den Prozess mit aufzunehmen und automatisierbar ist.

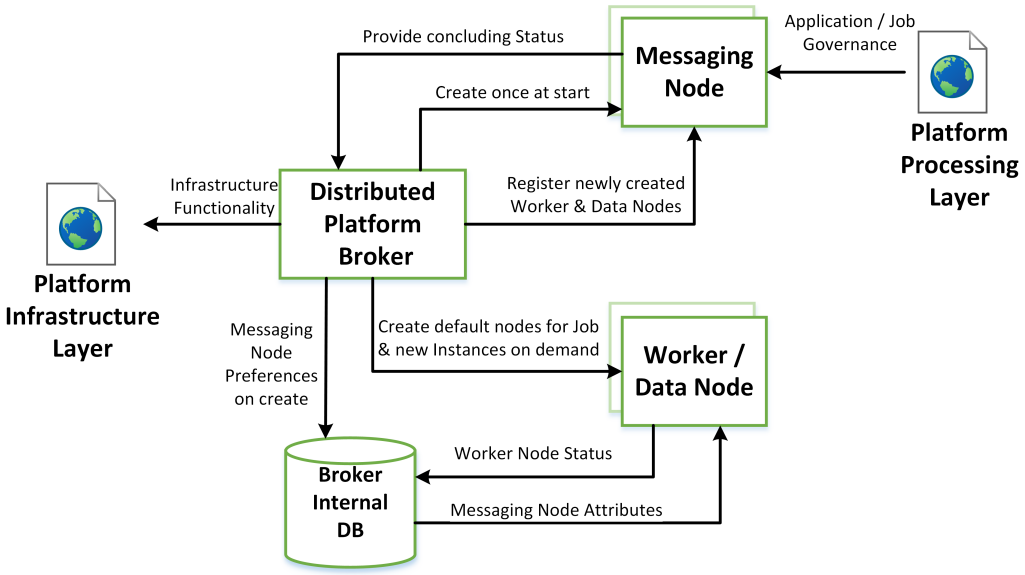

Abbildung 34.: Brokermechanismus für verteilte und dynamisch skallierbare Infrastrukturen

Alle zu Orchestrierung der Plattform bereitgestellten Funktionalitäten sind durch Webdokumente bzw. REST *calls* repräsentierbar. In diesem Szenario ist die Verarbeitung von Aufgaben auf die Anwendungen der verteilten Technologie abgestimmt. Die Funktionalität des jeweiligen *Job*s ist in einem projektspezifischen *Platform Processing Layer* zusammenzufassen. Die Koexistenz mehrerer verteilter Projekt-Plattformen innerhalb eines Hypervisor ist durch die Abbildung von Hierarchien innerhalb der *Broker*-internen Datenbank nicht ausgeschlossen. Infrastrukturmodifizierungen von Plattformen sind dem jeweiligen Administrator innerhalb einer grafischen Schnittstelle des *Distributed Platform Broker* zu Verfügung zur stellen.

Die Anpassung eines instanziierten *Cloud Image* an die momentane Orchestrierung einer Plattform ist eine andauernde Aufgabe. Abbildung 35 beschreibt durch die Manipulation der Infrastruktur ausgelöste Maintenance Abläufe eines instanziierten Plattformmoduls. Die Reaktion eines Hypervisor auf neu erstellte VMs beinhaltet verschiedene Parameter, welche von der *Distributed Platform Broker* Anwendung mit zugehörigen Technologieattributen persistiert wird. Ein, dem jeweiligen *Image* beigefügtes Instanziierungsskript wird innerhalb des ersten *Boot* Prozesses ausgeführt und verbindet sich nach dem befüllen der technologieabhängigen Konfigurationen mit der *Broker* Datenbank. Alle in diesem Moment aktiven Instanzen der verteilten Plattform sind dem neuen Knoten vor dem tatsächlichen Ausführen der Technologie initial bekannt zu geben.

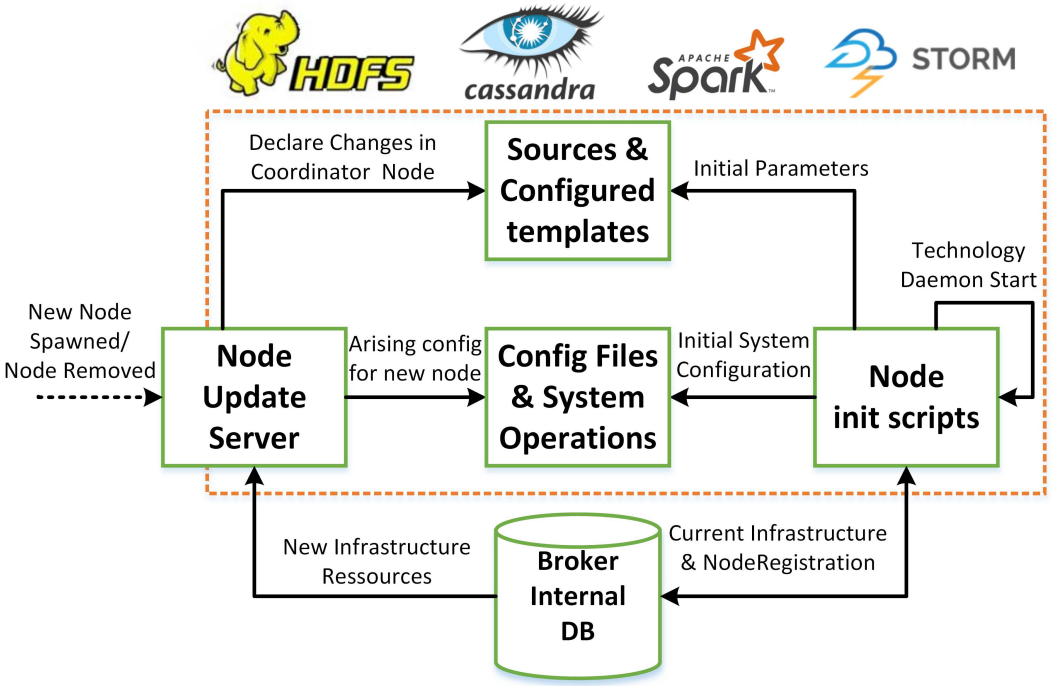

Abbildung 35.: Umsetzung der automatisierten Konfiguration von Modulen der verteilten Plattform

Bei dem Hinzufügen von Nodes in ein Cluster verlangt die unterliegende Technologie meist eine Neukonfiguration der Koordinationsserver. Ein innerhalb jeder Instanz kontinuierlich aktiver *Node Update Server* adaptiert Informationen von neuen Knoten und wendet entsprechende Aktionen der verwendeten Technologie an. Ein solches Ereignis entspringt sowohl dem hinzufügen, als auch dem entfernen einer Ressource der Plattform und basiert auf der durch den *Distributed Platform Broker* gepflegten Datenstruktur.

Diskussion

Im folgenden wird die beschriebene *Broker* Architektur am Beispiel von Hadoops ver-

teiltem Dateisystem HDFS diskutiert. Grundlegend basiert das HDFS auf mehreren *Data Nodes*, auf denen mehrere unveränderliche Datenblöcke durch einen *Name Node* platziert werden. Dieser koordiniert die im Cluster redundant vorgehaltenen Datenblöcke und ist für die Skalierung der Plattform zuständig. Die Anwendung der HDFS Technologie wird auf jedem Knoten durch identische *hadoop* Laufzeitumgebungen und Grundkonfigurationen gewährleistet. Innerhalb der vorgefertigten *Cloud Images* existiert jeweils ein *hduser* Account mit vollem Zugriff auf die *hadoop* Technologie. Während des ersten *Boot* Vorgangs dieser VMs Innerhalb einer *Data Node* ist die bestehende Konfiguration mit dem zugehörigen *hostname* der koordinierenden Instanz durch Interaktion mit der *Broker* Datenbank anzureichern. Bei der Initiierung einer *Name Node* kann diese Tätigkeit durch lokale Systemaufrufe umgesetzt werden.

Die Deklaration einer neuen *Data Node* muss sich über das gesamte Cluster erstrecken und kann durch den *Broker* mit direkten aufrufen an die Instanzen oder der Verwendung von Auslieferungstechnologien wie *Puppet*, getätigt werden. Zur Auflösung von beteiligten Instanzen eines Clusters wird die jeder Zeit auf allen Nodes aktuell gehaltene Datei */etc/hosts* verwendet. Die Konfiguration einer neuen *Data Node* ist mit dem Eintrag des *hostname* in der *$HADOOP_HOME/conf/slaves* und dem Starten des *hadoop-daemon* abgeschlossen.

Alle in Abbildung 35 abgebildeten Technologien sind auf die beschriebene Vorgehensweise anwendbar. Dieses Anwendungsbeispiel stellt keinen innovativen Ansatz der Orchestrierung von Ökosystemen dar, sondern soll die generischen Einsatzmöglichkeiten des Brokers unterstreichen. Für einen Anwendungsfall mit derartigen Anforderungen wird zu darauf spezialisierten und produktreifen Angeboten, wie beispielsweise *Elastic MapReduce*, geraten [WS14].

6.4. Verzicht auf die Definition von Datenquellen

Die in Kapitel 5 beschriebene Architektur bezieht sich zu großen Teilen auf den Umgang mit den flexiblen Eingangsdaten von virtualisierten Verarbeitungsplattformen. In den in Kapitel 3 aufgezeigten traditionellen Brokersystemen ist die Flexibilität der in die Vermittlungsschicht eingehenden Daten meist auf den Inhalt einer durch den Benutzer getätigten Anfrage beschränkt. Im folgenden soll die generische Komponente der vorgestellten Architektur anhand des Verzichts von auf einer Datenquelle beruhende Eingangsdaten verdeutlicht werden.

6.4.1. Vermittlung auf physische Maschine

Mit der Bearbeitung komplexer Aufgaben durch professionelle Anwendungen, können die Anforderungen an bestimmte Ressourcen einer Maschine steigen. Diese für den Betreib einer Applikation notwendigen Eigenschaften sind nicht in jedem Fall durch

ein CMS virtualisierbar. Um eine Menge an leistungsstarken physischen Maschinen auch netzwerkbasiert einzusetzen, bedarf es einer Orchestrierung der Zugriffe. Die in Abschnitt vorgestellten Gedanken zum Betrieb eines virtuellen Labors werden für das folgende Anwendungsbeispiel wieder aufgegriffen.

Problemstellung

Ein virtuelles Labor soll im Kontext einer Universität zur entfernten Umsetzung von mit spezieller Software zu bearbeiteten Problemstellungen für Studenten beitragen. Dabei ist die Sammlung an Anwendungen des Labors zweitrangig, vielmehr soll die direkte Bearbeitung von Aufgaben mit physischen und nicht virtualisierten Ressourcen einer Maschine, wie beispielsweise speziellen Grafikkarten oder eigens entwickelte Platinen, ausgenutzt werden. Sind diese Labormaschinen von mehreren Anwendern gleichzeitig nutzbar, können sich nicht zuletzt ökonomische Vorteile ergeben. Die in einem privaten Netzwerk aktiven Maschinen sind mit VPN Diensten der Universität bzw. Fakultät erreichbar. Um die Überlastung einer speziellen Maschine durch darauf aktiven Verbindungen zu vermeiden, ist ein Broker mit der Vermittlung von Anfragen auf physischen Labormaschinen betraut. Dabei kann die Vermittlungsschicht mit der Platzierung des jeweiligen Labors in Form von IP-Adressbereichen konfiguriert werden.

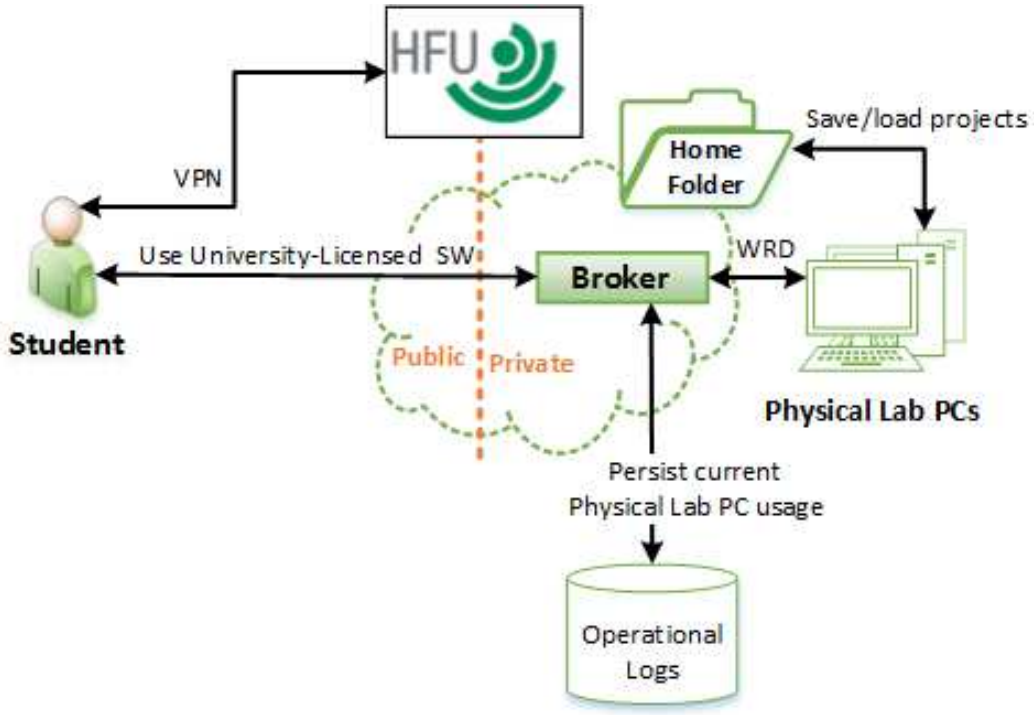

Abbildung 36.: Broker Technologie für Physikalische Maschinen

Mit der Heterogenität von Betriebssystemen der Studenten muss eine möglichst geeignete *Remote Desktop* Technologie gewählt werden. Um die Anwendbarkeit von

Remote Sessions in der eigenen Laufzeitumgebungen zu gewährleisten, sind meist technische Hürden zu meistern. Mit dem Einsatz von Produkten wie *ThinVNC* wird eine HTML5-basierte Verbindung zu einem entfernten Desktop ermöglicht. Diese *cross-browser*- und *cross-platform*-Anwendung kann also zur Lösung dieser, durch den Studenten zu bewältigenden, Teilprobleme beitragen.

Ist die Vermittlungsschicht mit der Benutzerverwaltung einer Universität verknüpft, kann die Authentifizierung des Studenten an der physischen Maschine des Labors auch automatisch anhand der bestehenden VPN Verbindung arangiert werden. Eine alternative zu dieser Vorgehensweise sind auf den Labormaschinen definierte Benutzerkonten, die dem Broker bekannt gemacht werden müssen.

Die momentane Auslastung der Maschinen durch offene Verbindungen ist in einer Broker-interne Datenstruktur abzubilden. Eine Konfiguration von *Quotas* über Verbindungen zu den einzelnen Maschinen muss auf die jeweilige Software und den Ressourcenverbrauch bei derer maximalen Anwendung abgestimmt sein. Ist der erwartete Andrang durch die momentan verfügbaren Ressourcen nicht zu befriedigen, kann ein derartiges Labor durch horizontale oder vertikale Skalierung erweitert werden. Die in der Datenbank einer Vermittlungsschicht gehaltenen Zustände eines Labors sind außerdem die Grundlage für das *Load Balancing*. Je nach definierter Benutzerschnittstelle eines Brokers kann diese Logik transparent gehalten werden. Dabei verwendete Algorithmen werden im einfachsten Fall auf die vorherrschenden Verbindungen aus der Datenbank angewandt. Da die Maschinen nicht durch ein CMS verwaltet werden, muss für die Berücksichtigung der tatsächlichen Rechenlast einer Ressource jedoch auf von traditionellen Monitoringsystemen gesammelte Informationen zurückgegriffen werden.

Die durch einen 'Connect' *Button* abstrahierte Vermittlungslogik verbindet den Studenten dynamisch auf einer ausgewählte Maschine mit freien Ressourcen. Mit der Anwendung der jeweiligen Applikationen entstehende Projektdaten oder sonstige Informationen sind daher in einem dem Benutzer zugehörigen Netzwerkordner abzulegen. Auf diese Weise sind dem Anwender auch bei Neuverbindungen durch den Broker auf eine andere Maschine alle bisher bearbeiteten Dateien zugänglich.

Diskussion

Einem Dozenten sind damit verschiedene Möglichkeiten zur Bereitstellung einer virtuellen Arbeitsumgebung gegeben. Die auf in einer Vermittlungsschicht verwiesenen, Maschinen können ebenfalls in unterschiedliche Kurse, bzw. darauf auszuführende Anwendungen, unterteilt werden. So können Maschinen, welche Applikationen zur Erarbeitung einer bestimmten Problemlösung bereitstellen, für einen festgelegten Zeitraum zur Verfügung gestellt werden. Der anwendende Student hat in diesem Fall

beispielsweise die Möglichkeit der Abgabe von Ausarbeitungen in einem auf der jeweiligen Maschine zugänglichen Netzwerkordner. Besonders interessant ist diese Art der Vermittlung bei Fernstudiengängen.

Steigt der Bedarf nach Verbindungen auf die bereitgestellten Maschinen kurzfristig, ist die Skalierung dieser Infrastruktur manuell zu bewerkstelligen. Das Vorhalten von Ausweichmaschinen kann je nach Lizenzmodell der beherbergenden Applikationen mit hohen Kosten verbunden sein. Ebenfalls ist die Zweckentfremdung der Maschinen, abhängig von der gewählten *Remote Desktop* Technologie, nicht auszuschließen

6.4.2. Auditierung verteilter Infrastrukturen und Dienste

Projektspezifische Infrastrukturen und Plattformen sind nicht immer während deren Instanziierung komplett konfigurierbar. Jedoch besteht die Möglichkeit der Definition unterschiedlicher VM-Images, deren von einander abhängigen Anwendungen automatisch verknüpft werden können. Durch die Definition der Infrastruktur einer Service Architektur aus auf einander abstimmbaren Grundmodulen, sind technische Details einer zu erstellenden verteilten Plattform mit Eingabemasken abstrahierbar. Die Haupttätigkeit des *Broker*s besteht daher aus der Auswahl und dem Starten des passenden *Cloud Image* und dem anschließenden Füllen von Konfigurationsdateien mit Parametern, welche durch den Administrator gewählt und vom CMS dynamisch vergeben werden.

Unter diesem Gesichtspunkt lassen sich verschiedene Anwendungsfälle durch eine abstrahierte Konfiguration des unterliegenden Software-Ökosystems beschleunigen.

Problemstellung

Hängt der Erfolg einer Unternehmung zu einem großen Maß von *Accountability*-Eigenschaften ab, ist der Einsatz von Technologien, wie dem in [RPR15] beschrieben Audit Agent System (AAS) oder vergleichbaren Agentensystemen ([RRR15]), denkbar. Dieses auf der Java Agent DEvelopement (JADE)-Technologie basierenden Agentensysteme ist in der Lage, verschlüsselt in verteilten Umgebungen zu kommunizieren. Das in Abbildung 37 gezeigte Anwendungsszenario einer Vermittlungsschicht bietet Administrationen eine Möglichkeit zur Definition einer auditierbaren Infrastruktur. Dabei liegt das Hauptaugenmerk auf der Sammlung von Informationen der Umsetzung von Unternehmungen in einer Infrastruktur.

Grundlegend definiert der *Auditor* einer verteilten Plattform die Umsetzung von Abgleichungen der ist-Zustände mit denen in einem Regelwerk festgehaltenen Anforderung. Dieses hat meistens einen direkten Bezug zu Komponenten eines Dienstes. Beispielsweise ist die Kommunikation in einer spezifischen Dienstkomponente nur verschlüsselt vorhergesehen. Ein mit der Analyse von Logdateien einer VM gefundener Regelverstoß gibt konkreten Informationen, wie den verwendeten IP Adressbereich

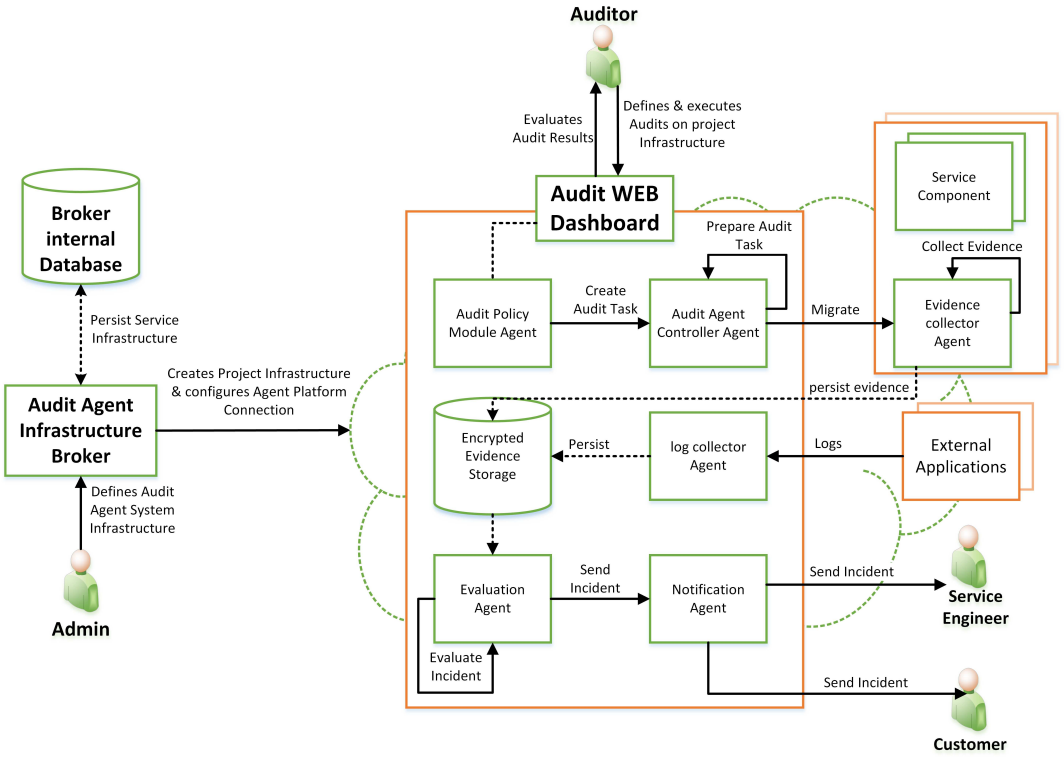

Abbildung 37.: Brokermechanismus für die Erstellung von auditierbaren Infrastrukturen

oder auf die jeweilige Dienstkomponente bezogene Details, automatisch an einen *Service Engineer* weiter. Bei vorheriger Definition von Benachrichtigungen sind diese Ereignisse auch dem Kunden des Dienstes zugänglich zu machen. Die Analyse von bereitgestellten Informationen einer externen Applikationen kann ebenfalls zu einer solchen Situation beitragen.

Eine zentrale Komponente ist mit der Orchestrierung von Audits auf der verteilten Infrastruktur betraut. Die in diesem AAS-*Core* platzierte Benutzerschnittstelle hat Kenntnis über die aus den Regelwerken ableitbaren *Audit*s. Mit der Registrierung von auf beteiligten Maschinen des Dienstes angesiedelten AAS-*Client* Prozessen bei der AAS-*Core* Komponente, wird ein namentlich ansprechbarer Kommunikationskanal zur administrativen Seite des Systems geschaffen. Wird ein neuer *Audit Task* im *Web Interface* definiert, findet die Instanziierung und Konfiguration von Softwareagenten innerhalb des *Audit Agent Controller* Moduls statt. Die Migration dieses Agenten zu einem sogenannten JADE-container ist zustandsbehaftet und benötigt auf der Zielmaschine angekommen keinerlei nachfolgende Konfiguration. Ein durch diesen *Evidence Collecting Agent* erkannte Regelverstöße werden innerhalb des auf dem AAS-*Core* platzierten *Encrypted Evidence Storage* persistiert. Mit der darauffolgenden Evaluierung des Ereignisses soll die Plausibilität und Ernsthaftigkeit eines Regelverstoßes, vor der Benachrichtigung von menschlichen Aktoren bestätigt werden.

Mit dem AAS empfehlen sich zwei *Cloud Image*s, welche jeweils die JADE Bibliotheken, sowie die für *Core* und *Client* grundlegenden AAS Anwendung beinhaltet. Innerhalb des Dashboards verschiebt sich der Fokus von variablen Eingangsdaten auf die Formulierung eines verteilten Systems. Der Anteil von während der Instanziierung eines *Cloud Image* durchgeführten Aktionen basiert dabei auf den Eingaben des Administrators und bezieht sich hauptsächlich auf die der AAS-*Core* VM beizufügenden *Policies*. Mit der Definition eines AAS-Instanznamen wird diese VM für *Client*s des Systems ansprechbar gemacht. Das Füllen der Hauptkonfigurationsdatei des AAS-*Core* hat ebenfalls Auswirkungen auf die verfügbaren *Audit Tasks*. Sind Fremdsysteme, wie CMS oder Datenbanken in eine solche Aufgabe involviert, benötigen die *Evidence Collecting Agent*s entsprechende Legitimierungen. Vor der Migration zu einer entfernten Laufzeitumgebung kann der jeweilige Agent durch den *Audit Agent Controller* mit den in der zentralen Konfigurationsdatei abgelegten Anmeldedaten von Fremdsystemen für die definierte Aufgabe bestückt werden. Auf der *Client*seite ist die Bekanntgabe des "MainContainer" eines AAS Systems anhand dessen Hostname oder IP-Adresse durch den Broker während der Instanziierung der VM dynamisch zu deklarieren. Der "ContainerName" des AAS-*Clients* assoziiert dabei die Laufzeitumgebung der JADE Komponenten mit einer physischen Maschine und dient ebenfalls dem grundlegenden Management, sowie der Orchestrierung von Agenten durch den Anwendungscode.

Diskussion

Prinzipiell vereinfacht dieser Broker das *Deployment* einer verteilten Infrastruktur, die auf jedem Knoten bereits mit einem vorkonfigurierten und vernetzten Agentensystem ausgestattet ist. Bei der Erstellung einer solchen Infrastruktur kann der Administrator über die initiale Anzahl der Maschinen entscheiden. Da sich neue *Client* VMs anhand einer Konfiguration dynamisch in ein AAS integrieren lassen, ist das Gesamtsystem vollautomatisiert um weitere Maschinen horizontal skalierbar.

Die Implementierung von den Komponenten eines Dienstes ist für die darauf angepasste Beweissicherung in jedem Fall auf den entsprechenden Knoten zu installieren. Bei einer neuen Version dieses JADE-basierten Systems, in der beispielsweise neue Agententypen hinzugefügt oder grundlegende Datenstrukturen geändert wurden, sind alle involvierten Maschinen manuell auf den selben Stand zu bringen. Mit dem automatischen Starten der AAS-Komponenten während dem Hochfahren einer VM, sind diese durch die *Core* Komponente direkt ansprechbar und befähigt die empfangenen Softwareagenten in der eigenen Laufzeitumgebung auszuführen. Führen szenariobedingte Umstände eines Agenten zu dessen Abbruch, kann die Anwendung eines Erlang Supervisor oder der Einsatz von beispielsweise *Daemon Tools* zur Gewährleistung des Dauerbetriebs eines AAS beitragen.

7. Zusammenfassung

Das folgende Kapitel gibt einen abschließenden Überblick der Hauptaspekte der vorliegenden Arbeit. Der darauffolgende Ausblick soll einige weiterführende Arbeiten und Bedingungen für den Produktiveinsatz der vorgestellten Vermittlungsschichten aufzeigen.

7.1. Fazit

Mit der Prototypischen Implementierung der in Kapitel 5 beschriebenen Architektur soll eine technologieunabhängige, modulare und erweiterbare Architektur für generische Vermittlungsdienste in cloudbasierten Umgebungen realisiert werden. Die Verarbeitung von Datensätzen wird durch damit konfigurierbaren Verarbeitungsplattformen umgesetzt. Die Erstellung einer solchen Vermittlungsschicht und den darin angebotenen Plattformen kann in wenigen Zeilen der vorgestellten DSL formuliert werden. Die individuelle Logik der damit formulierten Vermittlungsschicht, ist anschließend an gekennzeichneter Stelle des generierten Grundgerüstes zu implementieren.

Die Präparierung der zu instanziierenden Plattformen mit problemspezifischer Software geschieht dabei durch ebenfalls generierte Skripte. Mit dem Hinzufügen von Softwarepaketen und der Manipulation von Dateien dieses *Cloud Image*, stehen dem Entwickler des Brokers darin alle Operationen des gewählten Betriebssystems zur Verfügung. Damit sind dem Abbild alle Mechanismen zum Setzen von Konfigurationsparametern der Plattform, sowie für die Verarbeitung der Datenquelle benötigten Applikationen, bereit zu stellen.

Das Hauptaugenmerk der vorliegenden Arbeit ist die Effizienzsteigerung bei der Definition von Vermittlungsschichten. Der Anwendungsmodellierer dieses Brokers benötigt ausführliches Wissen über die jeweiligen Datenquellen, sowie von den Verarbeitungstechniken benötigte Parameter zur Initialisierung der involvierten Applikationen. Eine darauf aufbauende Benutzerschnittstelle abstrahiert komplexe Mechanismen im Bezug auf die Anwendung von Funktionalität eines CMS und der Weitergabe von Attributen der Eingangsdaten an die instanziierte Plattform. Dieser zentrale Ort einer problemspezifischen Unternehmung kann ebenfalls zur Weiterleitung an virtuelle Desktops der Plattformen verwendet werden. Mit der steigenden Informationsdichte von Datenquellen, bieten cloudbasierte Workflows, im Gegensatz zu traditionellen Verarbeitungsumgebungen, besonders interessante Vorzüge. Neben der Entlastung von

Ressourcen eines Endanwenders trägt diese cloudbasierte Bearbeitung von Problemstellungen auch zur geteilten Sicht auf Plattformen bei.

Die vielfältigen Einsatzgebiete dieser Art von Brokersystemen wurde anhand mehrerer Beispiele aufgezeigt. Mittels der generischen DSL sollte damit für das weite Anwendungsspektrum von damit formulierten Anwendungen sensibilisiert werden.

Eine derartig automatisierte Brokerstruktur trägt nicht aktiv zu einem bewussten Umgang mit Unternehmensressourcen bei. Auf der anderen Seite ist die manuelle Erstellung von Verarbeitungsinfrastrukturen auf Grund der menschlichen Komponente ein fehleranfälliger Vorgang. Die Schulung von Personal auf die einzelnen, für die Erstellung einer Verarbeitungsplattform benötigten Arbeitsschritte kann jedoch, abhängig von dem Wissensstand der Mitarbeiter, mit hohen Kosten verbunden sein.

7.2. Ausblick

Der Aufbau der gezeigten Architektur, sowie der darauf bezogenen DSL, ermöglicht eine Vielzahl an denkbaren Erweiterungen. Abhängig von der jeweiligen Problemstellung ist jedoch der Mehrwert einer solchen Vermittlungsschicht zu evaluieren. So sind beispiesweise vorgehaltene VM-Pools für *Load Balancing* Techniken nicht im allgemeinen sinnvoll. Die Anwendung eines derartigen Brokersystems in *Community Cloud*-basierten Umgebungen erfordert von den involvierten CSPs gemeinsam angewandte Regelwerke und die Registrierung der selben Plattform-*Image*s. In einem solchen Szenario ist die Einhaltung der formulierten Richtlinien, zur Erschaffung einer gemeinsamen Vertrauensbasis, regelmäßig zu auditieren. Der in Kapitel 5 vorgestellte Ablauf zur Erstellung einer Vermittlungsschicht kann als *proof of concept* angesehen werden und geht nicht auf die Sicherheit von derartigen Systemen ein. So ist die Kombination von ausgeführter Brokerfunktionalität mit geschäftsinternen Abläufen auf mögliche Schwachstellen und die Einhaltung von Regelwerken zu prüfen. Ein in diesem System verwendeter cloudbasierter Dienst kann ebenfalls auf die Schnittstellen einer Menge anderer Dienste zurückgreifen. Beispielweise gelten, abhängig von dem Standort eines Rechenzentrums im Vergleich zum Ursprungsdienst neben möglicherweise unterschiedlichen rechtlichen Grundlagen, auch andere SLAs. Neben diesem Sicherheitsaspekt sind insbesondere die *Connector*-Module solcher Plattform-Brokersysteme zu evaluieren.

Die Kombination mehrerer Plattform-Broker und die damit ermöglichten gegenseitigen Verwendung von Vermittlungsfunktionalitäten, ist nicht ausgeschlossen. Mit der Aggregation dieser Systeme können neue Dienstarten formuliert werden, die bei entsprechend konfigurierbaren *Cloud Images* von involvierten Brokern in der Erstellung eines Ökosystems zur Verarbeitung von Datensätzen resultieren.

Die logische Brokerarchitektur ist um beliebige Module erweiterbar. Eine tatsäch-

liche Anwendung von architekturfremden Modulen ist jedoch von der grundlegenden Technologie, die zur Verknüpfung der Komponenten einer Vermittlungsschicht verwendet wird, abhängig. Die Erweiterung der Architektur um einen zentralen *Thrift* Dienst, der für derartige Zusatzfunktionalität verwendet wird, ist ein denkbarer Lösungsansatz. Sind Zusatzmodule mit der DSL deklarierbar, bleibt die Formulierung solcher Broker weiterhin generisch. Die implementierten *Templates* der DSL sind technologiespezifisch und beziehen sich auf das *Django* Rahmenwerk. Eine Anwendung anderer Technologien verlangt tiefes Verständnis über diese, sowie über Strukturen der DSL und den Umgang mit dem *XTend* Framework.

Die Implementierung von automatisch ausgeführter Funktionalität ist nach momentanem Stand des Brokers durch manuelle Eingriffe, in aus dem Anwendungsmodell generierten und nicht für eine Manipulation vorhergesehenen Module, umzusetzen. Das Anreichern der DSL mit neuer Semantik muss in jedem Fall auch in den verwendeten *Templates* berücksichtigt werden. Die interne Datenbank eines Brokers wurde in der gezeigten DSL nicht berücksichtigt. Die manuelle Einbettung von Technologien für die Datenhaltung von instanziierten Plattformen, ergibt daher einen erhöhten Implementierungsaufwand. Mit einer für spezielle Technologien erweiterten DSL sind ebenfalls vorgefüllte *Connector* Module für die CMS-Anbindung, sowie auf die eingehenden Datenquellen angepasste Komponenten generierbar.

Der Endanwender dieses Brokers hat keinerlei Einfluss auf die Definition des für die Verarbeitungsplattform verwendeten *CLoud Image*s. Die von *lubguestfs* bereitgestellten Bibliotheken, wie der *python-guestfs*, können beispielsweise innerhalb eines *Django*-Broker direkt verwendet werden. Mit einem auf die zur Verfügung stehenden Funktionen des *python-guestfs* angepassten HTML-basierten Formular, steht einem Benutzer die Formulierung einer individuellen Verarbeitungsplattform frei. Wird die OpenStack Technologie als CMS verwendet, sind ebenfalls Orchestrierungsfunktionalitäten mit *Heat* und *Ceilometer* denkbare Erweiterungen der Benutzerschnittstelle.

Literaturverzeichnis

[AC13] Annenkov, D. V. ; Cherkashin, E. A.: Generation technique for Django MVC web framework using the stratego transformation language. In: *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2013, S. 1084–1087

[AC16] Abderrahim, Wiem ; Choukair, Zied: PaaS Dependability Integration Architecture Based on Cloud Brokering. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2016 (SAC '16). – ISBN 978–1–4503–3739–7, 484–487

[AFG+10] Armbrust, Michael ; Fox, Armando ; Griffith, Rean ; Joseph, Anthony D. ; Katz, Randy ; Konwinski, Andy ; Lee, Gunho ; Patterson, David ; Rabkin, Ariel ; Stoica, Ion ; Zaharia, Matei: A View of Cloud Computing. In: *Commun. ACM* 53 (2010), April, Nr. 4, 50–58. http://dx.doi.org/10.1145/1721654.1721672. – DOI 10.1145/1721654.1721672. – ISSN 0001–0782

[AGS15] Anchalia, Prajesh P. ; Gupta, Pracheta ; Shetty, Jyoti: A Customized Dashboard for VM Provisioning Using OpenStack. In: *7th International Conference on Computational Intelligence, Communication Systems and Networks, CICSyN 2015, Riga, Latvia, June 3-5, 2015*, 2015, 177–182

[AIN+16] Alsina, J. ; Iturriaga, S. ; Nesmachnow, S. ; Tchernykh, A. ; Dorronsoro, B.: Virtual Machine Planning for Cloud Brokering Considering Geolocation and Data Transfer. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016, S. 352–359

[AK14a] Amshavalli, R. S. ; Kavitha, G.: Increasing the availability of cloud resources using broker with semantic technology. In: *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*, 2014, S. 1578–1582

[AK14b] Awasthi, C. ; Kanungo, P.: Client Requirement Modeling Using Resource Broker Architecture in Cloud Computing Environment. In: *2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2014, S. 1–6

[AK15] Awasthi, C. ; Kanungo, P.: Resource allocation strategy for cloud computing environment. In: *2015 International Conference on Computer, Communication and Control (IC4)*, 2015, S. 1–5

[ASK07] Agarwal, Aditya ; Slee, Mark ; Kwiatkowski, Marc: Thrift: Scalable Cross-Language Services Implementation / Facebook. Version: 4 2007. http://thrift.apache.org/static/files/thrift-20070401.pdf. 2007. – Forschungsbericht

[BCF+12] Bellavista, P. ; Carella, G. ; Foschini, L. ; Magedanz, T. ; Schreiner, F. ; Campowsky, K.: QoS-aware elastic cloud brokering for IMS infrastructures. In: *2012 IEEE Symposium on Computers and Communications (ISCC)*, 2012. – ISSN 1530–1346, S. 000157–000160

[Bet13] Bettini, Lorenzo: *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. – 1–96 S. – ISBN 1782160302, 9781782160304

[Bil17] Bilodeau, Olivier: *Introducing Malboxes: a Tool to Build Malware Analysis Virtual Machines*. GoSecure. http://gosecure.net/2017/02/16/introducing-malboxes-a-tool-to-build-malware-analysis-virtual-machines/. Version: 2 2017. – visited 28.08.2017

[BKNT11] Baun, Christian ; Kunze, Marcel ; Nimis, Jens ; Tai, Stefan: *Cloud Computing: Web-Based Dynamic IT Services*. 1st. Springer Publishing Company, Incorporated, 2011. – 5–10, 15–22, 25–34 S. – ISBN 3642209165, 9783642209161

[BMR07] In: Bauer, Bernhard ; Müller, Jörg P. ; Roser, Stephan: *A Decentralized Broker Architecture for Collaborative Business Process Modelling and Enactment*. London : Springer London, 2007. – ISBN 978–1–84628–714–5, 115–125

[BPSN04] Baratto, Ricardo A. ; Potter, Shaya ; Su, Gong ; Nieh, Jason: MobiDesk: Mobile Virtual Desktop Computing. In: *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*. New York, NY, USA : ACM, 2004 (MobiCom '04). – ISBN 1–58113–868–7, 1–15

[Bra12] Braun, Jörg: *Der wolkige Desktop*. FreeX, 5 2012. – http://www.cul.de/data/freex52012pr.pdf

[CCCS14] Callegati, F. ; Cerroni, W. ; Contoli, C. ; Santandrea, G.: Performance of Network Virtualization in cloud computing infrastructures: The OpenStack case. In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, 2014, S. 132–137

[CFP+15] Corradi, A. ; Foschini, L. ; Pernafini, A. ; Bosi, F. ; Laudizio, V. ; Seralessandri, M.: Cloud PaaS Brokering in Action: The Cloud4SOA Management Infrastructure. In: *2015 IEEE 82nd Vehicular Technology Conference (VTC2015-Fall)*, 2015, S. 1–7

[CLPB16] Chamorro, L. ; López-Pires, F. ; Barán, B.: A Genetic Algorithm for Dynamic Cloud Application Brokerage. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, S. 131–134

[CML14] Chen, Min ; Mao, Shiwen ; Liu, Yunhao: Big Data: A Survey. In: *Mobile Networks and Applications* 19 (2014), Apr, Nr. 2, 171–209. http://dx.doi.org/10.1007/s11036-013-0489-0. – DOI 10.1007/s11036–013–0489–0. – ISSN 1572–8153

[COA+14] In: Chatterjee, Tamojit ; Ojha, Varun K. ; Adhikari, Mainak ; Banerjee, Sourav ; Biswas, Utpal ; Snášel, Václav: *Design and Implementation of an Improved Datacenter Broker Policy to Improve the QoS of a Cloud*. Cham : Springer International Publishing, 2014. – ISBN 978–3–319–08156–4, 281–290

[cow16] *Ninenines - Cowboy HTTP Server Website*. https://github.com/ninenines/cowboy/. Version: November 2016. – visited 26.09.2017

[CPJ16] Chauhan, S. S. ; Pilli, E. S. ; Joshi, R. C.: A broker based framework for federated Cloud environment. In: *2016 International Conference on Emerging Trends in Communication Technologies (ETCT)*, 2016, S. 1–5

[CR15] Carlo Rengo, Harm D.: Teleporting virtual machines / System and Network Engineering, TNO. 2015. – Forschungsbericht. – https://www.os3.nl/_media/2014-2015/courses/rp1/p14_report.pdf

[CT09] Cesarini, Francesco ; Thompson, Simon: *ERLANG Programming*. 1st. O'Reilly Media, Inc., 2009. – 152–154 S. – ISBN 0596518188, 9780596518189

[Dat13] Datastrax: *Comparing the Hadoop Distributed File System (HDFS) with the Cassandra File System (CFS)*. https://www.datastax.com/resources/whitepapers/hdfs-vs-cfs, 2013. – Whitepaper

[DG08] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *Commun. ACM* 51 (2008), Januar, Nr. 1, 107–113. http://dx.doi.org/10.1145/1327452.1327492. – DOI 10.1145/1327452.1327492. – ISSN 0001–0782

[DVS+12] Deboosere, Lien ; Vankeirsbilck, Bert ; Simoens, Pieter ; De Turck, Filip ; Dhoedt, Bart ; Demeester, Piet: Efficient resource management for virtual desktop cloud computing. In: *The Journal of Supercomputing* 62 (2012), Nov, Nr. 2, 741–767. http://dx.doi.org/10.1007/s11227-012-0747-0. – DOI 10.1007/s11227–012–0747–0. – ISSN 1573–0484

[EL14] Elman, Julia ; Lavin, Mark: *Lightweight Django*. 1st. O'Reilly Media, Inc., 2014. – ISBN 149194594X, 9781491945940

[Ent15] Enterprise, Hewlett P.: *Become a cloud service broker Seven success factors when transforming IT into a cloud service broker*. https://www.hpe.com/h20195/v2/GetPDF.aspx/4AA5-1140ENN.pdf, November 2015

[ER16] Estrada, Raul ; Ruiz, Isaac: *Big data SMACK – A Guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka*. Berkeley, CA,; New York : Apress, 2016 https://www.amazon.de/Big-Data-SMACK-Apache-Cassandra/dp/1484221745. – ISBN 9781484221754 1484221753

[FFG+14] Fifield, Tom ; Fleming, Diane ; Gentle, Anne ; Hochstein, Lorin ; Proulx, Jonathan ; Toews, Everett ; Topjian, Joe: *OpenStack Operations Guide*. 1st. O'Reilly Media, Inc., 2014. – 1–14, 55–62, 157–170 S. – ISBN 1491946954, 9781491946954

[fos] *Foss Cloud Website*. http://www.foss-cloud.de/wiki/Hauptseite. – visited 26.09.2017

[GA16] Gupta, M. K. ; Annappa, B.: Trusted partner selection in broker based cloud federation. In: *2016 International Conference on Next Generation Intelligent Systems (ICNGIS)*, 2016, S. 1–6

[gar09] *Gartner Says Cloud Consumers Need Brokerages to Unlock the Potential of Cloud Services*. http://www.gartner.com/newsroom/id/1064712, 2009. – visited 30.08.2017

[Geo16] George, N.: *Mastering Django: Core*. Packt Publishing, Limited, 2016. – 2, 5–104, 233–271, 344–356 S. https://books.google.de/books?id=71dHMQAACAAJ. – ISBN 9781787281141

[GGB+15] Guzek, M. ; Gniewek, A. ; Bouvry, P. ; Musial, J. ; Blazewicz, J.: Cloud Brokering: Current Practices and Upcoming Challenges. In: *IEEE Cloud Computing* 2 (2015), Mar, Nr. 2, S. 40–47. http://dx.doi.org/10.1109/MCC.2015.32. – DOI 10.1109/MCC.2015.32. – ISSN 2325–6095

[Heb13] Hebert, Fred: *Learn You Some Erlang for Great Good!: A Beginner's Guide*. San Francisco, CA, USA : No Starch Press, 2013. – 3, 263–280 S. – ISBN 1593274351, 9781593274351

[HJK+09] In: Heidenreich, Florian ; Johannes, Jendrik ; Karol, Sven ; Seifert, Mirko ; Wende, Christian: *Derivation and Refinement of Textual Syntax for Models*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – ISBN 978–3–642–02674–4, 114–129

[HNK+16] Haroon, T. ; Neena, S. ; Krishnaprasad, K. K. ; Wilson, R. ; Simon, S. ; Martin, J. P.: Convivial private cloud implementation system using OpenStack. In: *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016, S. 3484–3488

[Inc13] Inc, Bromium: *Micro-virtualization vs Software Sandboxing*. https://www.bromium.com/sites/default/files/Bromium-Whitepaper-Micro-virtualization-vs-sandboxing_0.pdf, 2013. – Whitepaper, Bromium-WP-0905

[JUIE15] Joe-Uzuegbu, C. K. ; Iwuchukwu, U. C. ; Ezema, L. C.: Application virtualization techniques for malware forensics in social engineering. In: *2015 International Conference on Cyberspace (CYBER-Abuja)*, 2015, S. 45–56

[Jö13] Jörges, Sven: *Lecture Notes in Computer Science*. Bd. 7747: *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*. Springer, 2013. – 3–221 S. – ISBN 978–3–642–36126–5

[KCC+10] Ku, Kyong-I ; Choi, Won-Hyuk ; Chung, Moonyoung ; Kim, Kiheon ; Kim, Won-Young ; Hur, Sung-Jin: Method for distribution, execution and management of the customized application based on software virtualization. In: *2010 The 12th International Conference on Advanced Communication Technology (ICACT)* Bd. 1, 2010. – ISSN 1738–9445, S. 493–496

[KFF+08] Keahey, Kate ; Figueiredo, Renato ; Fortes, Jos ; Freeman, Tim ; Tsugawa, Maurcio: Science clouds: Early experiences in cloud computing for scientific applications. In: *Cloud computing and applications* 2008 (2008), S. 825–830

[Khe15] Khedher, Omar: *Mastering OpenStack*. Packt Publishing, 2015. – 1–37, 103–140, 275–296 S. – ISBN 1784395641,9781784395643

[KHK+14] Kim, Heejae ; Ha, Yoonki ; Kim, Yusik ; Joo, Kyung-No ; Youn, Chan-Hyun: A VM Reservation-Based Cloud Service Broker and Its Performance Evaluation. In: Leung, Victor C. M. (Hrsg.) ; Lai, Roy X. (Hrsg.) ; Chen, Min (Hrsg.) ; Wan, Jiafu (Hrsg.): *CloudComp* Bd. 142, Springer, 2014 (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering). – ISBN 978–3–319–16049–8, 43-52

[KJB15] Khanna, P. ; Jain, S. ; Babu, B. V.: BroCUR: Distributed cloud broker in a cloud federation: Brokerage peculiarities in a hybrid cloud. In: *International Conference on Computing, Communication Automation*, 2015, S. 729–734

[KKR+12] Kim, Seong-Hwan ; Kang, Dong-Ki ; Ren, Ye ; Park, Yong-Sung ; Joo, Kyung-No ; Youn, Chan-Hyun ; Park, YongSuk: An Experimental Cloud Resource Broker System for Virtual Application Control with VM Allocation Scheme. In: *7th International Conference on Ubiquitous Information Technologies & Applications (CUTE), Hong Kong*, 2012

[KL16] Kong, W. ; Luo, Y.: Multi-level image software assembly technology based on OpenStack and Ceph. In: *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, 2016, S. 307–310

[LA12] Langmann, R. ; Arts, S.: Application virtualization in virtual learning labs. In: *2012 9th International Conference on Remote Engineering and Virtual Instrumentation (REV)*, 2012, S. 1–4

[LA14] Lachgar, M. ; Abdali, A.: Generating Android graphical user interfaces using an MDA approach. In: *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, 2014. – ISSN 2327–185X, S. 80–85

[Lan04] Lang, Carsten: *Organisation der Software-Entwicklung*. Wiesbaden : Deutscher Universitätsverlag, 2004. – 201–358 S. http://dx.doi.org/10.1007/978-3-663-01622-9_4. http://dx.doi.org/10.1007/978-3-663-01622-9_4. – ISBN 978–3–663–01622–9

[LBR16] Letrache, Khadija ; Beggar, Omar E. ; Ramdani, Mohamed: Modeling and creating KPIs in MDA approach. In: *CIST*, IEEE, 2016, S. 222–227

[LCX12] Liu, Jianxun (Hrsg.) ; Chen, Jinjun (Hrsg.) ; Xu, Guandong (Hrsg.): *2012 Second International Conference on Cloud and Green Computing, CGC 2012, Xiangtan, Hunan, China, November 1-3, 2012*. IEEE, 2012 . – ISBN 978–1–4673–3027–5

[LEN+14] Lethrech, M. ; Elmagrouni, I. ; Nassar, M. ; Kriouile, A. ; Kenzi, A.: Domain Specific Modeling approach for context-aware service oriented systems. In: *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, 2014, S. 575–581

[LTM+12] Liu, Fang ; Tong, Jin ; Mao, Jian ; Bohn, Robert ; Messina, John ; Badger, Lee ; Leaf, Dawn: *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology (Special Publication 500-292)*. USA : CreateSpace Independent Publishing Platform, 2012. – ISBN 1478168021, 9781478168027

[MG11] Mell, Peter M. ; Grance, Timothy: SP 800-145. The NIST Definition of Cloud Computing. Gaithersburg, MD, United States : National Institute of Standards & Technology, 2011. – Forschungsbericht

[MJ08] Malich, S. ; Jung, P.D.S.E.P.D.R.: *Qualität von Softwaresystemen: Ein pattern-basiertes Wissensmodell zur Unterstützung des Entwurfs und der Bewertung von Softwarearchitekturen*. Gabler Verlag, 2008 (Gabler Edition Wissenschaft). – ISBN 9783834910301

[MMTR16] Mayer, Ruben ; Mayer, Christian ; Tariq, Muhammad A. ; Rothermel, Kurt: GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. New York, NY, USA : ACM, 2016 (DEBS '16). – ISBN 978–1–4503–4021–2, 309–316

[nis13] *NIST Cloud Computing Security Reference Architecture*. https://www.nist.gov/document-4641, 2013

[nis17] *NIST Cloud Management Broker (working document)*. https://www.nist.gov/itl/61-cloud-management-broker. Version: 2017. – visited 26.09.2017

[OY16] Ossi Ylijoki, Jari P.: Perspectives to Definition of Big Data: A Mapping Study and Discussion. In: *Journal of Innovation Management* JIM4, 1 (2016), 69-91. http://www.open-jim.org/article/view/198

[Pil09] Pilgrim, Mark: *Dive Into Python 3*. Berkely, CA, USA : Apress, 2009. – 336 S. – ISBN 1430224150, 9781430224150

[PMG16] Pacini, E. ; Mateos, C. ; Garino, C. G.: Broker Scheduler based on ACO for Federated Cloud-based scientific experiments. In: *2016 IEEE Biennial Congress of Argentina (ARGENCON)*, 2016, S. 1–7

[PYS11] Prof. Yunlin Su, Prof. Song Y. Yan (.: *Principles of Compilers: A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag Berlin Heidelberg, 2011. – 101–107 S. – ISBN 978–3–642–20834–8,978–3–642–20835–5

[Rak15] Rakowski, Krzysztof: *Learning Apache Thrift*. Packt Publishing, 2015. – 1–15, 50–71, 88–95 S. – ISBN 1785882740, 9781785882746

[RDAC16] Roy, D. G. ; De, D. ; Alam, M. M. ; Chattopadhyay, S.: Multi-cloud scenario based QoS enhancing virtual resource brokering. In: *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*, 2016, S. 576–581

[RPR15] Rübsamen, Thomas ; Pulls, Tobias ; Reich, Christoph: Secure Evidence Collection and Storage for Cloud Accountability Audits. In: *CLOSER 2015 - Proceedings of the 5th International Conference on Cloud Computing and Services Science, Lisbon, Portugal, 2015*, 2015, S. 321 – 330

[RRR15] In: Ruf, Philipp ; Rübsamen, Thomas ; Reich, Christoph: *Agent-Based Evidence Collection in Cloud Computing*. Cham : Springer International Publishing, 2015. – ISBN 978–3–319–17199–9, 185–198

[SBBS12] Spillner, Josef ; Brito, Andrey ; Brasileiro, Francisco V. ; Schill, Alexander: A Highly-Virtualising Cloud Resource Broker. In: *IEEE Fifth International Conference on Utility and Cloud Computing, UCC 2012, Chicago, IL, USA, November 5-8, 2012*, 2012, 233–234

[Sha13] Shaw, Zed: *Learn Python The Hard Way*. 3rd. Addison-Wesley, 2013. – 206–211 S. – ISBN 0321884914, 978–0321884916

[SV06] Stahl, Thomas ; Völter, Markus: *Model-Driven Software Development: Technology, Engineering, Management*. Chichester, UK : Wiley, 2006. – 11–29, 55–71, 82, 85–116, 143–179, 379–391, 396 S. – ISBN 978–0–470–02570–3

[SXFD16] Su, Z. ; Xu, Q. ; Fei, M. ; Dong, M.: Game Theoretic Resource Allocation in Media Cloud With Mobile Social Users. In: *IEEE Transactions on Multimedia* 18 (2016), Aug, Nr. 8, S. 1650–1660. http://dx.doi.org/10.1109/TMM.2016.2566584. – DOI 10.1109/TMM.2016.2566584. – ISSN 1520–9210

[VBD+13] Voelter, Markus ; Benz, Sebastian ; Dietrich, Christian ; Engelmann, Birgit ; Helander, Mats ; Kats, Lennart C. L. ; Visser, Eelco ; Wachsmuth, Guido: *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. – 142– 161 S. http://www.dslbook.org. – ISBN 978–1–4812–1858–0

[VGO+16] Villamizar, Mario ; Garces, Oscar ; Ochoa, Lina ; Castro, Harold ; Salamanca, Lorena ; Verano, Mauricio ; Casallas, Rubby ; Gil, Santiago ; Valencia, Carlos ; Zambrano, Angee ; Lang, Mery: Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* 00 (2016), S. 179–182. http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/CCGrid.2016.37. – DOI doi.ieeecomputersociety.org/10.1109/CCGrid.2016.37

[vir17] *virt-builder - Build virtual machine images quickly*. http://libguestfs.org/virt-builder.1.html. Version: 2017. – visited 26.09.2017

[Wol15] Wolff, E.: *Microservices: Grundlagen flexibler Softwarearchitekturen*. Dpunkt.Verlag GmbH, 2015. – 314–316 S. https://books.google.de/books?id=SrccswEACAAJ. – ISBN 9783864903137

[WS14] In: Wadkar, Sameer ; Siddalingaiah, Madhu: *Hadoop in the Cloud*. Berkeley, CA : Apress, 2014. – ISBN 978–1–4302–4864–4, 343–356

[YL16] Yang, J. ; Li, G.: Earth Observation data integration and opening system. In: *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2016, S. 5481–5484

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Freiburg, den 31.08.2017 Philipp Ruf

A. Anwendung der Broker DSL

Das folgende Kapitel bezieht sich auf das in Abbildung 32 gezeigte Anwendungsszenario aus Kapitel 6. Die beispielhafte Anwendung dieses Szenarios soll die benutzerseitige Anwendung der Vermittlungsschicht demonstrieren, wobei keine Rücksicht auf die Sinnhaftigkeit der eingehenden Daten gelegt wurde.

Ausformulierung der Broker DSL

Im folgenden ist die ausformulierte DSL-Instanz des *ShowcaseBroker* zu betrachten.

```
1  CloudResourceBroker ShowcaseBroker {
2    DashboardMode standalone //creating a stanalone Django application
3    isThriftBasedBroker //using Apache Thrift for action execution
4    CloudManagementConnection openStackConnection {
5      cmsConnection userAuthentication  //a user must authenticate in broker dashboard with OpenStack using its credentials
6      ConfigFile openStackConfig{"hostname" "port"} //Configuration file contains DNS/IP and port to OpenStack nova-compute API
7    }
8    BrokerTask showcaseTask{
9      DataConnection mysqlConnection {ConfigFile mysqlConnectionConfic{"hostIP" "hostPort" "username" "password" "database"}} //declaring MySQL connection
10     DataConnection cassandraConnection {ConfigFile cassandraConnectionConfig{"hostIP" "hostPort" "username" "password" "database"}} //declaring Cassandra connection
11     DashboardModel showcasePredeployment {
12       Table MySQLMetadataTab {
13         tableName "relationalDataTable"
14         Column id{"id" "primaryKey"}
15         Column attribute{"someAttribute" "Some Attribute"}
16         Column comment{"someComment" "Some Comment"}
17         Column dataLocation{"dataLocation" "" isHiddenColumn}
18         action createInstanceAction (create onDataAttribute id buttonText "Create Platform for relational Data") //action dependent on primary key of row
19         connector mysqlConnection //references MySQL connection
20         UserData relationalUserData {
21           script (onStartCommand "echo 'export MOUNT_PATH=  ' >> /etc/profile.d/mountPointDeclaration.sh" columnRefference dataLocation) //persistent environment variable
22         }
23       }
24       Table CassandraMetadataTab {
25         tableName "columnOrentedDataTable"
26         Column id{"id" "primaryKey"}
27         Column attribute{"someAttribute" "Some Attribute"}
28         Column triggerComment{"someComment" "Some Comment"}
29         Column dataLocation{"dataLocation" "" isHiddenColumn}
30         action createInstanceAction (create onDataAttribute id buttonText "Create Platform for column-oriented Data") //action dependent on primary key of row
31         connector cassandraConnection //references Cassandra connection
32         UserData relationalUserData {
33           script(onStartCommand "echo 'export MOUNT_PATH=  ' >> /etc/profile.d/mountPointDeclaration.sh" columnRefference dataLocation ) //persistent environment variable
34           script(onStartCommand "sed -i.bkp '$ i processingApplicationLocation=/opt/comunOrientedApp/' /etc/someOtherTool/audit.cfg") //alter configuration file of different tool in Platform
35         }
36       }
37       Table AggregatedMetadataTab {
38         tableName "aggregationDataTable"
39         Column id{"id" "primaryKey"}
40         Column mysqlAttribute{"mysqlAttribute" "MySQL Attribute"}
41         Column cassandreAttribute{"cassandreAttribute" "Cassandra Attribute"}
42         Column aggregatedValue{"aggregatedValue" "Meaningful Aggregation Value"}
43         Column comment{"someComment" "Some Comment"}
44         Column mySqlDataLocation{"mySqlDataLocation" "" isHiddenColumn}
45         Column cassandraDataLocation{"cassandraDataLocation" "" isHiddenColumn}
46         action createInstanceAction (create onDataAttribute id buttonText "Create Platform for aggregated Data")
47         connector mysqlConnection //references MySQL connection
48         connector cassandraConnection //references Cassandra connection
49         UserData relationalUserData {
50           script(onStartCommand "echo 'export MYSQL_MOUNT_PATH=  ' >> /etc/profile.d/mysqlMountPointDeclaration.sh"  columnRefference mysqlAttribute) //persistent environment variable
51           script(onStartCommand "echo 'export CASSANDRA_MOUNT_PATH=$' >> /etc/profile.d/cassandraMountPointDeclaration.sh"  columnRefference cassandreAttribute) //persistent environment variable
52         }
53       }
54       Table AggregatedPlatformInstancesTab {
55         tableName "aggregatedPlatformInstancesTab"
56         Column instanceID{"instanceid" "Instance ID" }
57         Column instanceSocket{"instancesocket" "Physical Instance Socket" representsInstanceSocket} //IP of instances
```

```
58          Column platformType{"platformType" "Platform Type"}
59          Column status{"status" "Instance Status" isFeedbackColumn(workerConnectionAttribute instanceSocket
                )}  //gets platform status by value of 'instanceSocket' column
60          action vncActionOnInstance (noVNC onDataAttribute instanceID buttonText "VNC Connection") //
                connect to instance GUI or CMD in browser
61          action deleteActionOnInstance(delete onDataAttribute instanceID buttonText "Delete Instance")
62          action customActionOnPlatform(custom onDataAttribute instanceID buttonText "Snapshot and Backup")
                //a custom process triggered on button click
63          instanceTable //declared as instance table - this data structure will be persisted in broker
                internal DB (apart from the dynamic 'status' column)
64        }
65      }
66      WorkerImage relationalImage {
67        distribution "ubuntu-14.04"
68        processingApplication RelationalAnalysis {
69          startApplicationCommand "cd /opt/relationalApp; ./startRelationalAnalysis.sh /opt/relationalData"
                //takes data provision as argument
70          producingLogFile "relationalAnalysisLog.out"
71          DynamicDataProvision dProvision {
72            mountCommand "sshfs -i ~/.ssh/sharedKey $MOUNT_PATH /opt/relationalData" //must provide key in
                  WorkerImage and set dynamic $MOUNT_PATH in user_data
73          }
74        }
75      }
76      WorkerImage columnOrientedImage {
77        distribution "centos-7.2"
78        processingApplication ColumnOrientedAnalysis {
79          startApplicationCommand "cd /opt/comunOrientedApp; ./startcolumnOrientedAnalysis.sh"
80          producingLogFile "columnOrientedAnalysisLog.out"
81          DynamicDataProvision dProvision {oneTimeDefintion ""}
82        }
83      }
84      WorkerImage aggregatedImage {
85        distribution "ubuntu-16.04"
86        processingApplication AggregatedAnalysis {
87          startApplicationCommand "./~/startAggregatedAnalysis.sh"
88          producingLogFile "aggregatedAnalysisLog.out"
89          DynamicDataProvision dProvision {
90            mountCommand "sshfs -i ~/.ssh/sharedKeyMySQL $MYSQL_MOUNT_PATH /opt/relationalData" //must
                  provide key in WorkerImage and set dynamic $MYSQL_MOUNT_PATH in user_data
91            mountCommand "sshfs -i ~/.ssh/sharedKeyCassandra $CASSANDRA_MOUNT_PATH /opt/columnOrientedData"
                  //must provide key in WorkerImage and set dynamic $CASSANDRA_MOUNT_PATH in user_data
92          }
93          GuiDefinition aggregationGUI {
94            desktopPackage "lxde"
95            profilePlacement "~/"
96            desktopUser "lxdeUser"
97            guiAutoLogin
98          }
99        }
100     }
101   }
102   brokerDescription "Broker serving a WEB interface for creating Instances of relational, column oriented
          and aggregated analysis VMs"
103 }
```

Da es sich bei dieser Anwendung um einen *Thrift*-basierten Broker handelt(Zeile *3*), werden alle Aktionen als separate Dienste in entsprechenden IDL Dateien generiert. Die in den Einzelnen Tabellen platzierten Aktionen (Zeilen *18, 30, 46 & 60 - 62*) wurden dennoch nicht komplett verteilt ausformuliert, vielmehr sind Problemstellungen einer gemeinsamen Domäne mit einander verknüpft. Innerhalb des *thriftServiceConnector.py* Moduls befinden sich die für eine Orchestrierung auszuformulierenden Aufrufe der *Thrift* Dienste. Da in diesem Beispiel von einem einzelnen CMS System ausgegangen wurde, ist die *openStackConnection* Komponente (Zeile *4*) für alle Hypervisorfunktionalitäten zuständig. Im folgenden wird die Arbeitsweise der *deleteActionOnInstance* Funktionalität verdeutlicht.

```
1  in  table views.py AggregatedPlatformInstancesTabView:
2    on button "Delete Instance":
3      get affected instanceID
4      call deleteActionOnInstance_AggregatedPlatformInstancesTab with id and user OpenStack token
5      -> parse IP and Port of Thrif OpenStackConnection service implementation host
6         create Thrift client object
7         call remote 'deleteInstance' implementation
8         -> Service implementation receives instance ID and Token
9            executes deletion of instance by ID
10           returns boolean outcome to client
11           <-
12        interprets deletion result
13        removes instance from internal Instances, if successful
14        <-
15     returns action outcome to AggregatedPlatformInstancesTabView
16     <-
17   renders current internal DB content for AggregatedPlatformInstancesTab table
18 returns content to AggregatedPlatformInstancesTab.html
```

Bis auf die *customActionOnPlatform* Funktion wird sich daher der clientseitigen *getServiceClient_ OpenStackConnection* implementierung des *thriftServiceConnector.py*

Moduls bedient. Die aus den generierten *Django* modulen aufgerufenen *Thrift-Client*s beziehen sich jedoch auch auf die Verbindungen zu den Datenbanken (Zeilen *9 & 10*).

Aufbau der Benutzerschnittstelle des Brokers

Im folgenden wird die aus der DSL generierte Benutzerschnittstelle gezeigt.

Mit dem in Zeile *2* deklarierten *DashboardMode* wird die durch *Django* gepflegte Benutzerverwaltung in die Vermittlungsschicht mit aufgenommen. Abbildung 38 zeigt die unveränderte Anmeldemaske des *ShowcaseBroker*.

Abbildung 38.: Anmeldemaske des Showcase Broker

Bei erfolgreichem Anmelden an der Vermittlungsschicht wird auf dessen Willkommensseite vermittelt. Durch die in Zeile *5* deklarierte *userAuthentication* besitzt das Benutzermodell zusätzliche Felder für ein *OpenStack Token* und dessen Ablaufdatum. In Abbildung 39 ist das entsprechende Formular aufgezeigt, welches bereits ausgefüllt wurde und unter Kommunikation mit der *CloudManagementConnection* aus Zeile *4* die *djangTest* Benutzerattribute der Sitzung aktualisierte.

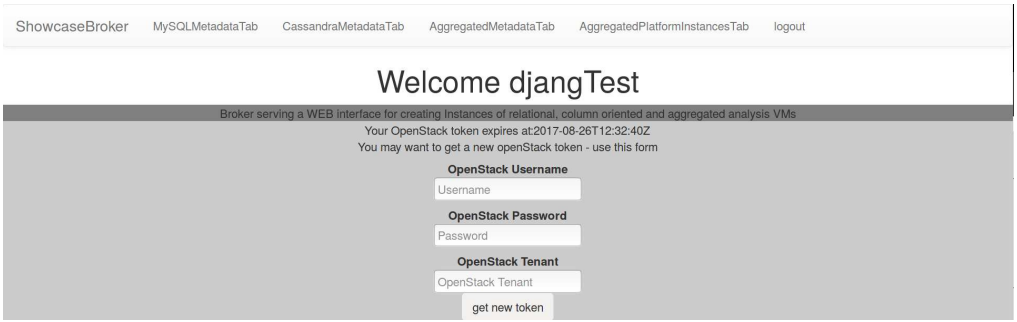

Abbildung 39.: Willkommensseite des Showcase Broker

Abbildung 40 zeigt die in den Zeilen *12, 24* und *37* deklarierten Tabellen in einem gemeinsamen Schaubild. Darin dargestellte Attribute von einzelnen Spalten besitzen dabei keine Aussagekraft. Die in Zeile *18* deklarierte *createInstanceAction* resultiert in einem *Button*, welcher die Kommunikation mit der *CloudManagementConnection* abstrahiert. Die in Zeile *17* formulierte *dataLocation* Spalte ist auf Grund des *isHiddenColumn* Ausdrucks hierbei für den Benutzer nicht sichtbar.

Mit der in Zeile *54* deklarierten *AggregatedPlatformInstancesTab* Tabelle werden Inhalte der durch die Vermittlungsschicht gepflegten Datenbank visualisiert. Die Darstellung der durch eine *create* Aktion erstellten Verarbeitungsplattformen ist mit den in Zeilen *60 - 62* definierten Aktionen verknüpft. Eine *vncActionOnInstance* Aktion erstellt unter Kommunikation mit der *CloudManagementConnection* eine durch OpenStack bereitgestellte VNC Verbindung her, auf die der Anwender weitergeleitet

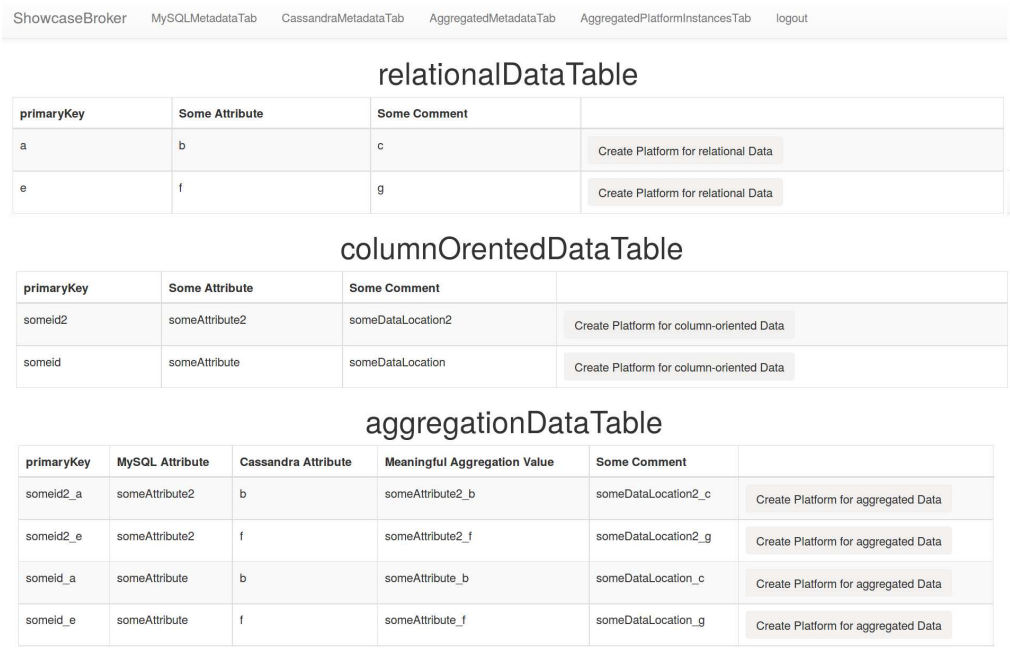

Abbildung 40.: Eingangsdaten des Showcase Broker

wird. Mit der *deleteActionOnInstance* Aktion kann der Benutzer die bereits erstellten Plattformen aus der OpenStack Umgebung entfernen. Die *customActionOnPlatform* ist mit der Generierung von Anwendungscode nicht automatisch mit weiteren Brokermodulen verknüpft und muss individuell ausformuliert werden.

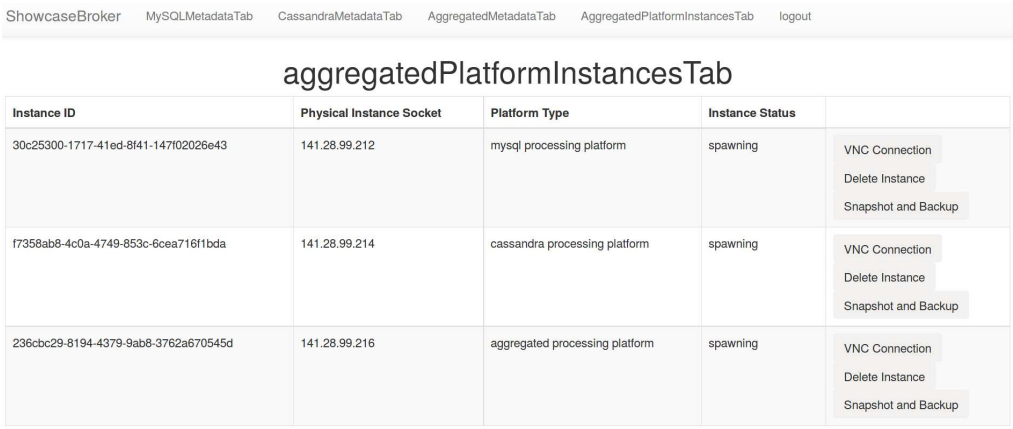

Abbildung 41.: Virtualisierte Verarbeitungsplattformen des Showcase Broker

Für eine ausführliche Betrachtung dieser Anwendung wird auf die beiliegende CD-ROM verwiesen. Mit der darin enthaltenen Implementierung soll die Arbeitsweise dieses Brokers verdeutlicht werden. Die Erstellung von bedeutungsvollen *Cloud Images* wurde dabei nicht berücksichtigt.

B. CD-Rom

- LaTeXCode
- Programmcode
  - Broker DSL
  - Django Templates
  - Anwendungsbeispiel
- Onlinequellen
- Sonstiges